Fachhochschule Düsseldorf

Fachbereich 3 - Elektrotechnik

Praxisprojekt

Untersuchung eines Mikroprozessor-Betriebssystems mit 6LoWPAN-Stack in Hinblick auf die Portierung auf neue Prozessoren

Analysis of an operation system for microprocessors containing a 6LoWPAN-Stack in terms of porting to other processors

Vorgelegt von: Mirco Kern
Matrikelnummer: 525698
Vorgelegt am: 07.01.2012
1. Prüfer: Herr Prof. Dr. Ulrich Schaarschmidt
2. Prüfer Herr Prof. Dr. Wolfgang Lux

Kurzfassung

Innerhalb dieser Dokumentation des im Labor für Informatik an der Fachhochschule Düsseldorf durchgeführten Parxisprojekt werden zunächst Grundlagen vermittelt, die nötig sind, um den Kontext des Projekts und das Kommunikationsprotokoll 6LoWPAN zu verstehen. Da das Ziel des Praxisprojekts die Untersuchung eines Kommunikationssystems mit 6LoWPAN-Stack hinsichtlich dessen Portierungsmöglichkeiten ist, werden hierauf verschiedene, bereits auf dem Markt befindliche Lösungen analysiert und verglichen, die entweder direkt den drahtlosen Datenaustausch mit Hilfe des 6LoWPAN-Protokolls oder aber die Erweiterung zu einem eigenen Stack ermöglichen und dargestellt, warum das Mikrocontroller-Betriebssystem Contiki als geeignetster Kandidat hierfür erscheint. Als Basis für die Analyse in Hinblick auf eine mögliche Portierung dient der Mikrocontroller ATmega128RFA1 des Herstellers Atmel, der über eine integrierte Transceiver-Einheit verfügt. Nach der Vermittlung der Grundlagen des Contiki-Betriebssystems, die insbesondere den schichtenartigen Aufbau der Verzeichnisse und das komplexe, über Makefiles organisierte Build-System betreffen, folgt die Beschreibung der Vorgehensweise bei der Portierung von Contiki und den dabei aufgetretenen Problemen. Da während der Arbeit an diesem Praxisprojekt eine neue Version des Contiki-Betriebssystems veröffentlicht wurde, wird zusätzlich auf diese Version und die vorhandenen Inkompatibilitäten zwischen den verschiedenen Contiki-Versionen eingegangen und eine Beispielanwendung für die neue Version implementiert. Die Performancemessungen verschiedener Kommunikationshardware runden diese Arbeit zusammen mit dem Fazit, das nocheinmal auf das Betriebssystem Contiki, den Mikrocontroller ATmega128RFA1 und mögliche Vertiefungs- und Optimierungsmöglichkeiten eingeht, ab.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Projektumfeld . . . 1
  - 1.1.1 Softwareentwicklung . . . 1
  - 1.1.2 Bisher genutzte Hardware-Komponenten . . . 2
- 1.2 IPv6 und 6LoWPAN . . . 2
  - 1.2.1 Warum IPv6? . . . 2
  - 1.2.2 6LoWPAN . . . 4
- 1.3 Aufgabenstellung . . . 5

2 Gegenüberstellung verfügbarer Kommunikationslösungen **6**
- 2.1 µracoli (µController Radio Communication Library) . . . 6
- 2.2 Atmel MAC (Media Access Control) . . . 7
- 2.3 µIPv6 (SICSlowpan) . . . 8
- 2.4 Atmel RUM (Route Under MAC) . . . 8
- 2.5 ContikiOS . . . 9
- 2.6 TinyOS . . . 10
- 2.7 FreeRTOS . . . 10
- 2.8 Für dieses Projekt genutzte Kommunikationslösung . . . 11
- 2.9 Weitere Stacks und Betriebssysteme . . . 12

3 Genutzte Hardware innerhalb des Praxisprojektes **13**
- 3.1 Das ATmega128RFA1 Evaluation Kit . . . 13
- 3.2 Kompatibilität des Controllers ATmega128RFA1 zu anderen Prozessoren . . . 14
  - 3.2.1 Nicht implementierte Ports . . . 14
  - 3.2.2 Externer Speicher . . . 14
  - 3.2.3 High Voltage Programming . . . 14
  - 3.2.4 Integrierte und externe Oszilatoren . . . 14
  - 3.2.5 Analoge Ein-/Ausgänge . . . 15
- 3.3 Nutzung der ATmega128RFA1 Transceiver-Einheit . . . 15
  - 3.3.1 Schnittstellen zwischen Transceiver-Einheit und Controller . . . 15
    - 3.3.1.1 Register-Zugriff . . . 15
    - 3.3.1.2 Zugriff auf den Frame-Buffer . . . 16
    - 3.3.1.3 Das Transceiver Pin Register . . . 16
    - 3.3.1.4 Interrupts . . . 17
    - 3.3.1.5 Steuerung des Transceiver-Zustandes . . . 18
    - 3.3.1.6 Identifikation der Transceiver-Einheit . . . 18

3.3.2 Ablauf der Datenübertragung . . . 18
3.3.2.1 Initialisierung und Konfiguration des Transceiver-Moduls 18
3.3.2.2 Empfang von Daten . . . 19
3.3.2.3 Versenden von Daten . . . 20
3.4 Erste Funktionsüberprüfung . . . 21

4 Grundlagen zu ContikiOS 22
4.1 Multi-Threading und Protothreads . . . 22
4.2 Events und Interprozesskommunikation . . . 22
4.3 Verzeichnisstruktur . . . 22
4.3.1 Das Anwendungsverzeichnis (apps) . . . 23
4.3.2 Das Systemverzeichis (core) . . . 23
4.3.3 Prozessorspezifischer Code (cpu) . . . 24
4.3.4 Dokumentation zu Contiki (doc) . . . 25
4.3.5 Beispielprojekte (examples) . . . 25
4.3.6 Plattformspezifischer Code (platform) . . . 25
4.3.7 Hilfsprogramme (tools) . . . 25
4.4 Das Contiki Build-System . . . 26
4.4.1 Nutzung des Programmes make unter Windows . . . 26
4.4.2 Aufbau der Makefiles . . . 27

5 Portierung von ContikiOS 2.4 30
5.1 Erstellen einer neuen Hardware-Plattform . . . 30
5.2 Anpassen der CPU-spezifischen Hardware-Abstraktionen und Treiber . . . 31
5.2.1 Die Registermap . . . 32
5.2.2 Der Kommunikationstreiber . . . 32
5.3 Portierung auf eine bisher nicht implementierte Prozessor-Familie . . . 34
5.4 Probleme . . . 34

6 Arbeiten mit ContikiOS 2.5 35
6.1 Kompatibilität zu älteren Contiki-Versionen . . . 35
6.1.1 Kompatibilität der Befehle . . . 35
6.1.2 Kompatibilität des Kommunikationsprotokolls . . . 36
6.2 Tests zur Datenübertragungsgeschwindigkeit . . . 36

7 Fazit / Ausblick 38

A Anhang 40
A.1 Listings . . . 40
A.2 Protokolle der Geschwindigkeitstests . . . 47

Abbildungsverzeichnis 49

Tabellenverzeichnis 50

Listingverzeichnis **51**

Literaturverzeichnis **52**

1 Einleitung

1.1 Projektumfeld

Diese im Labor für Informatik an der Fachhochschule Düsseldorf erarbeitete Projektarbeit steht im Zusammenhang mit dem WieDAS[1]-Projekt, das in Kooperation mit der Hochschule Rhein-Main und Vertretern aus Industrie und Wirtschaft durchgeführt und durch das Bundesministerium für Bildung und Forschung geförderter wird. Innerhalb des WieDAS-Projektes werden Grundlagen zu dem relativ jungen Themengebiet Ambient Assisted Living (AAL) erarbeitet, welches sich mit technischen Systemen beschäftigt, die es älteren oder in irgendeiner Weise körperlich eingeschränkten Personen erlauben, möglichst lange in ihrer gewohnten Umgebung zu verbleiben und ein eigenständiges Leben zu führen. Von verschiedenen Mitarbeitern des Labors für Informatik an der Fachhochschule Düsseldorf werden hierzu Soft- und Hardwaresysteme entwickelt, die sowohl medizinische Bereiche, als auch Lösungen, die in das Themengebiet der Hausautomatisierung fallen, abdecken. Neben der Funktion an sich sind bei der Entwicklung Offenheit, Erweiterbarkeit, Sicherheit, Selbstorganisation und Mobilität als wichtige Aspekte zu berücksichtigen[1]. Da die Akzeptanz der Nutzer für ein Gerät sinkt, sobald es ihn (nach außen) als hilfebdürftig erscheinen lässt, sollen sich die einzelnen Komponenten zudem möglichst unauffällig in die gewohnte Umgebung integrieren lassen, beziehungsweise von Außenstehenden nicht als Komponente eines Assistenzsystems erkannt werden.

1.1.1 Softwareentwicklung

Die Entwicklung der Anwendungs-Software geschieht im Labor für Informatik an der Fachhochschule Düsseldorf in der Programmiersprache Java, wobei die erstellten Programme auf dem OSGi[2]-Framework aufbauen. Zu dem grundsätzlichen Vorteil der Plattformunabhängigkeit von Java besitzen die auf Basis dieses Frameworks entwickelten RCP[3]-Anwendungen den Vorzug, dass sich einzelne Komponenten im laufenden Betrieb des Systems austauschen lassen, oder auch neue Komponenten hinzugefügt werden können, ohne den Betrieb zu unterbrechen. Zudem lassen sich innerhalb des Frameworks weitere Funktionen nutzen, die Möglichkeiten zur Implementierung von Selbstorganisation und Fehlertoleranz bieten, so dass softwareseitig die zuvor genannten Anforderungen an das Assistenzsystem erfüllt werden können.

[1] Wiesbaden-Düsseldorfer Ambient Assisted Living Service Platform
[2] Open Services Gateway initiative
[3] Rich Client Platform

1.1.2 Bisher genutzte Hardware-Komponenten

Die für das WieDAS-Projekt hergestellten Hardware-Komponenten der Sensorknoten bestehen neben weiteren elektronischen Bauteilen aus Mikrocontrollern des Herstellers Atmel, die vorzugsweise in der Programmiersprache *C* programmiert werden und einem ebenfalls im Labor für Informatik entwickeltem Funkmodul, das die drahtlose Kommunikation zwischen den einzelnen Komponenten ermöglicht. Da die überwiegende Mehrheit der Geräte unabhängig von einer kabelgebundenen Energieversorgung betrieben wird, ist es unerlässlich, dass sowohl die Hardware, als auch die auf den Mikrocontrollern laufende Software möglichst energiesparend konzipiert werden. Hierzu dient unter anderem auch der Einsatz des Übertragungsprotokolls 6LoWPAN[4], auf welches in Unterkapitel 1.2.2 noch gesondert eingegangen wird.

Um die anfallenden Sensordaten auszuwerten, beziehungsweise Aktoren, wie zum Beispiel den innerhalb des AAL-Projekts genutzten fernsteuerbaren Fensteröfner, steuern zu können, wird enweder direkt an dem auswertenden und steuernden Computersystem des Benutzers oder an einem entsprechend konfigurierten Router ein weiterer Transceiver benötigt. Im jetzigen Stadium des AAL-Projekts dient hierzu der Raven USB-Stick des Herstellers Atmel, der über eine geänderte Software die Kommunikation über das 6LoWPAN-Protokoll unterstützt. Da die Weiterentwicklung dieser Einheit nicht primär zu den Zielen des hier dokumentierten Praxisprojektes zählt, wird auf weitere Einlassungen an dieser Stelle verzichtet und auf entsprechende Dokumente des Herstellers Atmel [2] und die Bachelorthesis von Herrn Bernhard Esders [3] verwiesen, in der die zur Kommunikation benötigte Hard- und Software bereits behandelt wurde.

1.2 IPv6 und 6LoWPAN

1.2.1 Warum IPv6?

Das zur paketvermittelten Datenübertragung genutzte Internet Protocol Version 6 (IPv6), welches auch als Internet Protocol Next Generation (IPnG) bezeichnet wird, ist eine Weiterentwicklung des zuvor genutzten Protokolls in der Version 4 (IPv4). Es wurde bereits 1998 von der Internet Engineering Taskforce (IETF) standardisiert, konnte sich jedoch aus verschiedenen Gründen bis heute nicht vollständig durchsetzen.

Die Weiterentwicklung wurde nötig, da das Internet in immer neue Bereiche des alltäglichen Lebens vordringt, was zur Folge hat, dass immer mehr Geräte über IP-Adressen angesprochen werden müssen. Da die Adressierung innerhalb des IPv4-Protokolls durch die Adresslänge von 32 Bit auf eine Anzahl von 2^{32}, also etwa 4,3 Milliarden Adressen beschränkt ist, kommt es seit einiger Zeit bereits zu Ressourcenengpässen bei der Adressvergabe, welchen durch teilweise komplexe technische Maßnahmen begegnet werden muss. Ein Beispiel für die Nutzung derartiger Übergangstechnologien ist die Adressierung

[4] IPv6 over Low Power Wireless Personal Area Networks

von Mobilfunkgeräten. Da hierbei die Anzahl der genutzten Geräte die Ressourcen der durch den Anbieter vergebbaren Adressen bei weitem übersteigen würde, teilen sich in deutschen Mobilfunknetzen mehrere tausend Geräte eine IP-Adresse. Die individuelle Zuordnung der Adressen zu den jeweiligen Mobilfunkgeräten wird hierbei intern durch den Provider über verschiedene technische Maßnahmen geregelt. Ähnlich verhält es sich bei Servern im Internet, die zwar jeweils nur eine IP-Adresse besitzen, aber verschiedene Inhalte über unterschiedliche Internetseiten bereit stellen können. IPv6 begegnet dem Problem, indem die Adresslänge auf 128 Bit vergrößert wurde. Somit wurde die Anzahl der möglichen Adressen um den Faktor 2^{96} erhöht und entspricht nun ungefähr 340 Sextillionen ($340 \cdot 10^{36}$) Adressen.

Eine weitere Antriebsfeder für die Weiterentwicklung des IP-Protokolls war die unzureichende Eignung für den Einsatz innerhalb neuer Technologien, wie WebTV, Video-on-Demand oder Electronic Commerce, in denen die Dienstgüte (Quality of Service, kurz QoS) oder die Sicherheit der Kommunikationsverbindung eine Rolle spielt. Um den Anforderungen dieser Dienste gerecht zu werden, wurden die Kopfdaten (Header) der IP-Pakete um Funktionen erweitert, die zum Beispiel die gewünschte Dienstklasse oder Informationen zur Verschlüsselung von Paketen beinhalten. [4]

Die wichtigsten Verbesserungen können wie folgt zusammengefasst werden:

- Vergrößerung des Adressraums
- Erweiterung der Routing-Funktionen
- Vereinfachung und Verbesserung der Protokollinformationen
- Möglichkeit zur Nutzung von Unicast-, Anycast-, und Multicast-Datenströmen
- Einführung von Quality-of-Service (QoS)
- Unterstützung von Authentifizierung und anderen Sicherheitsmechanismen

Wie bereits weiter oben beschrieben, exististieren jedoch Hemnisse, welche die vollständige Einführung von IPv6 verhindern. Hierzu zählen neben einigen technischen Problemen auch die schon angesprochenen Übergangstechnologien, durch die eine starke Fokussierung auf die IPv6-Technologie durch Unternehmen wie auch Privatpersonen unnötig erscheint. Hinzu kommen diverse datenschutzrechtliche Bedenken. Diese basieren darauf, dass es mittels IPv6 prinizipiell möglich ist, jedem Gerät eine dauerhafte IP-Adresse zuzuweisen, über die Nutzer über unterschiedliche Netzwerke hinweg identifizierbar werden und somit eine Überwachung des Nutzungsverhaltens möglich ist. Dem entgegen stehen jedoch technische Maßnahmen, durch die ein Netzwerkgerät selbständig einen sich ändernden Geräteanteil der Netzwerkadresse erzeugen kann. Diese als Privacy Extensions bezeichneten Funktionen sind in vielen Betriebssystemen entweder standardmäßig aktiviert oder können zumindest durch den Nutzer des Systems nachträglich eingschaltet werden. [5, 6]

1.2.2 6LoWPAN

OSI-Referenzmodell			TCP/IP-Referenzmodell			6LoWPAN
Protokolle	Schichten	Nr.	Nr.	Schichten	IP-Protokolle	6LoWPAN-Protokolle
HTTP HTTPS FTP SMTP	Anwendung (Application)	7	4	Anwendung (Application)	HTTP HTTPS FTP SMTP	Anwendungs- protkolle
	Darstellung (Presentation)	6				
	Sitzung (Session)	5				
TCP UDP	Transport (Transport)	4	3	Transport (Transport)	TCP UDP	UDP ICMP
IPv4 IPv6	Vermittlung (Network)	3	2	Internet	IPv4 IPv6	IPv6
						LoWPAN
Ethernet Token Ring FDDI	Sicherung (Data Link)	2	1	Netzzugang (Data Link)	Ethernet MAC (Media Access Control)	IEEE 802.15.4 MAC (Media Access Control)
	Bitübertragung (Physical)	1			Ethernet PHY (Physical)	IEEE 802.15.4 PHY (Physical)

Abb. 1.1: OSI-Referenzmodell, TCP/IP-Referenzmodell und der 6LoWPAN-Stack im Vergleich

Das Kommunikationsprotokoll 6LoWPAN ist aus der Idee entstanden, das Internet Protocol auch relativ leistungsschwachen Geräten, wie zum Beispiel eingebetteten Systemen und Sensorknoten, zur Verfügung zu stellen. Hierzu bietet 6LoWPAN eine effiziente Möglichkeit der Datenübertragung über Netzwerke, die auf dem Standard IEEE[5] 802.15.4 basieren, der auch als LR WPAN[6] bezeichnet wird und auf dem neben 6LoWPAN auch ZigBee und weitere nicht kabelgebundene Datenübertragungsverfahren, beziehungsweise Netzwerkprotokolle, aufsetzen. In diesem Standard, der die unteren beiden Schichten des OSI[7]-Referenzmodell definiert, sind Übertragungsmechanismen festgelegt, die sich durch Eigenschaften wie geringe Leistungsaufnahme, sichere Datenübertragung und die Nutzung der lizenskostenfreien ISM[8]-Bänder insbesondere für drahtlose Sensornetzwerke und für direkt am Körper getragene Sensoren und Aktoren (Body Area Network, kurz BAN) eignen.

Allerdings bringt die Nutzung von IPv6 über LR WPAN einige Probleme mit sich, die darin begründet sind, dass zwei grundlegend verschiedene Netzwerktypen verbunden werden, die

[5]Institute of Electrical and Electronic Engineers
[6]Low Rate Wireless Personal Area Network
[7]Open Systems Interconnection
[8]Industrial, Scientific and Medical Band

unter anderem unterschiedlichle Paketgrößen verwenden. Weitere Unterschiede bestehen in der Größe des Frame Overheads, in dem die dem Paket zugehörigen Verwaltungsdaten abgelegt sind und in der Adressierung, die bei LR WPAN mit 64 Bit erfolgt, während Geräte in IPv6-Netzwerken, wie bereits im vorherigen Abschnitt angesprochen, mit 128 Bit adressiert werden. [7]

Abbildung 1.1 am Anfang dieses Kapitels verdeutlicht die Ähnlichkeit des Aufbaus des IP- und 6LoWPAN-Stacks. Einschränkungen des 6LoWPAN-Stacks gegenüber dem IP-Stack bestehen darin, dass die Vermittlung ausschließlich über das IP-Protokoll in der Version 6 erfolgt und aus Gründen der Leistungsfähigkeit und Komplexität zunächst auf eine verbindliche Implementierung des verbindungsorientierten TCP[9]-Protokolls verzichtet wurde [8]. Auf der Anwendungsebene besteht prinzipiell die Möglichkeit jegliche Protokolle zu nutzen, die nicht zwingend verbindungsorientiert arbeiten. Allerdings ist bei der Verwendung von 6LoWPAN und ähnlichen auf dem Standard IEEE 802.15.4 aufbauenden Protokollen zu berücksichtigen, dass diese nicht für eine hohe Datenübertragungsrate entwickelt wurden und somit für einige Anwendungen ungeeignet sind.

1.3 Aufgabenstellung

Im Labor für Informatik wurden bereits einige lauffähige Geräte entwickelt, die in der Lage sind, das 6LoWPAN-Protokoll für die Kommunikation zu nutzen. Diese Geräte basieren auf dem 8-Bit Mikrocontroller ATmega1284P, auf dem die eigentlichen Datenverarbeitungsaufgaben implementiert sind und dem Transceiver-Chip AT86RF230, über welchen die Daten drahtlos übertragen werden können.

Das dieser Dokumentation zugrunde liegende Praxisprojekt hat das Ziel, zu untersuchen wie die Kommunikation über 6LoWPAN mit anderen als den gegenwärtig genutzten Prozessoren und Transceiver realisiert werden kann. Hierzu soll beispielhaft der Controller ATmega128RFA1 verwendet werden, der wie die beiden zuvor angesprochenen Mikrochips von dem Hersteller Atmel stammt. Das Besondere an diesem Chip ist, dass durch die Integration des Funkmoduls in den Controller (System on a Chip) kein externer Transceiver benötigt wird. Hierdurch entsteht weniger Platzbedarf für die Schaltung auf der Platine, was besonders bei möglich platzsparend zu haltenden Modulen, wie zum Beispiel dem im WieDAS-Projekt zu entwickelnden Notfallknopf, als wünschenswert erachtet wird. Ein weiterer Vorteil liegt darin, dass sowohl die Steuerung des Transceiver-Einheit als auch die Übertragung der zu versendenden Daten an das Transceiver-Modul direkt über die Register des Controllers geschieht, was theoretisch die Möglichkeit einer schnelleren Datenübertragung in Verbindung mit einer höheren Energieeffizienz bietet.

[9] Transmission Control Protocol

2 Gegenüberstellung verfügbarer Kommunikationslösungen

In diesem Kapitel sollen einige bereits vorhandene Lösungen vorgestellt und verglichen werden, die möglicherweise eine Grundlage für die kabellose Datenübertragung über das 6LoWPAN-Protokoll darstellen. Dies können entweder Betriebssyteme für Mikrocontroller mit integrierter IPv6-Funktionalität sein oder auch von einem Betriebssystem unabhängige IPv6-Stacks. Wichtig ist hierbei, dass die Lösung frei von Lizenzkosten ist und auf möglichst viele verschiedene Hardware-Plattformen portiert werden kann, um die Offenheit des Systems gewährleisten zu können. Hierbei gilt als Minimalanforderung die Portierbarkeit oder direkte Lauffähigkeit auf verschiedene Prozessoren des Herstellers Atmel, da zu diesem Zeitpunkt ausschließlich diese in dem bereits vorgestellten AAL-Projekt genutzt werden.

2.1 µracoli (µController Radio Communication Library)

Das µracoli-Paket stellt eine unter modifizierter BSD[1]-Lizenz stehende Bibliothek dar, die auf einfache Weise eine Datenübertragung unter Verwendung von IEEE 802.15.4-fähigem Transceivern erlaubt. Hierzu stellt das µracoli-Paket diverse low-level Treiber-Routinen für den Betrieb des Transceiver-Chips bereit, zu denen unter anderem die Konfiguration des Übertragungskanals, das Senden und Empfangen von Frames und die Kollisionsvermeidung über das CSMA/CD[2]-Verfahren gehören. Desweiteren bietet das µracoli-Paket verschiedene Hilfsfunktionen an, um die Erstellung neuer Anwendungen zu erleichtern. Dies sind zum Beispiel Funktionen für die Timer-Steuerung, zur Kontrolle der GPIO-Register[3] oder für die Vereinfachung der seri-

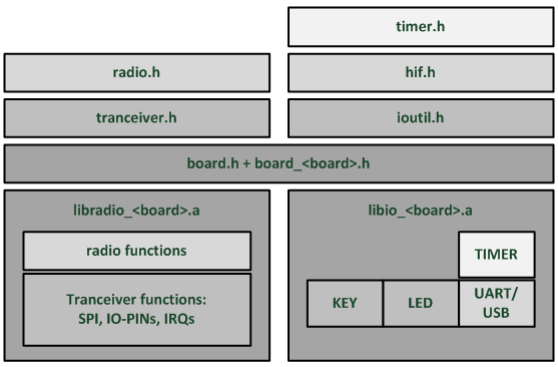

Abb. 2.1: Struktur des µracoli-Pakets

[1] Berkeley Software Distribution
[2] Carrier Sense Multiple Access/Collision Detection
[3] General Purpose Input/Output Register

ellen Kommunikation über UART[4]. Somit ist der Programmierer in der Lage, mit relativ einfachen Mitteln Datenrahmen (Frames) über die Funkschnittstelle auszutauschen. Da durch µracoli lediglich die untere Schicht des TCP/IP-Referenzmodells abgebildet wird, bietet das µracoli-Paket jedoch keinerlei Unterstützung für Protokolle höherer Schichten. Daher wären für die Einbindung in ein 6LoWPAN-Netzwerk noch zusätzlich Implementierungen nötig. Eventuell ist es möglich hierfür Teile anderer TCP/IP-Stacks werden, die auf das µracoli-Paket aufsetzen könnten.

Die bereitgestellten Funktionen sind zunächst für die Verwendung von Controllern des Herstellers Atmel vorgesehen, lassen sich aber durch das in µracoli implementierte Konzept der Hardware- beziehungsweise Plattform-Abstraktion auch für die Verwendung mit Controllern anderer Hersteller portieren. In *Abbildung 2.1* ist die Struktur des µracoli-Pakets und das Zusammenspiel der Include-Dateien (zum Beispiel radio.h) mit den µracoli-Bibliotheken (zum Beispiel libradio_<board>.a) ersichtlich. [9]

2.2 Atmel MAC (Media Access Control)

Mit der frei verfügbaren MAC-Stack Software bietet Atmel eine Referenzimplementierung für Transceiver zur Nutzung des Übertragungsstandards IEEE 802.15.4 an. Hierbei unterstützt der Stack Peer-to-Peer-Kommunikation in sternförmigen Topologien und kann sowohl mit Baecons als auch mit Non-Beacon-Netzwerken umgehen. *Abbildung 2.2* zeigt die Architektur des Stacks, wobei der Anschaulichkeit halber einige für das Verständnis nicht benötigte Module nicht abgebildet wurden. Wichtig ist jedoch, dass der Stack, ähnlich wie das µracoli-Paket, lediglich die untere Schicht des TCP/IP-Referenzmodells, also die Netzzungangsschicht, abdeckt. Daher bietet der MAC-Stack keinerlei Routing-Funktionalitäten und es können ausschließlich einfache, nach IEEE 802.15.4 standardisierte Datenpakete übertragen werden. Dies entspricht zwar noch nicht der gewünschten Kommunikation über 6LoWPAN, ist jedoch unter Umständen als Grundlage für weitere Implementierungen nutzbar. Hierzu müssten auf den vorhandenen Stack zusätzliche Protokolle aufgesetzt werden, die zumindest die rudimentäre Übertragung von IPv6-Datenpaketen ermöglichen.

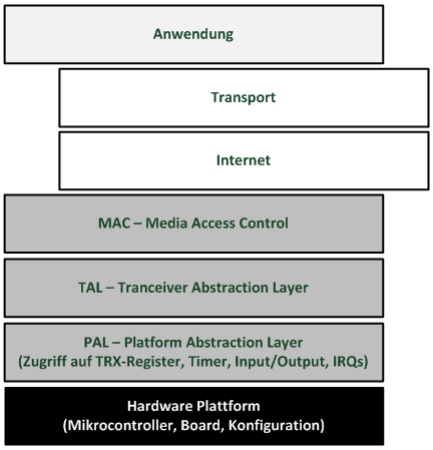

Abb. 2.2: Architektur des MAC-Stacks

Vorteilhaft ist, dass die einzelnen Schichten der MAC-Stack Software leicht zu nutzende APIs[5] beinhalten, über die ihre Funktionen von höheren Ebenen angesprochen werden

[4] Universal Asynchronous Receiver Transmitter
[5] Application Programming Interface

können und dass durch Veränderung der Hardware-Abstraktionen prinzipiell auch eine Portierung für Mikroprozessoren anderer Hersteller möglich ist. Der im Praxisprojekt verwendete Prozessor ATmega128RFA1 wird in der hier untersuchten Version 2.7.1 der Software (letztes Update im Oktober 2011) bereits unterstützt. [10]

2.3 µIPv6 (SICSlowpan)

µIPv6 ist einer der kleinsten durch das IPv6 Forum[6] zertifizierten IPv6-Stacks. Der Stack wurde als open-source Projekt veröffentlicht und darf somit unter Einhaltung gewisser Vorschriften privat und auch kommerziell frei verwendet werden. Durch seinen geringen Speicher- und Ressourcenbedarf eignet er sich insbesondere für den Einsatz in Mikrocontroller-basierten Sensornetzwerken. Der modulare Aufbau des µIPv6-Stacks ermöglicht dessen Verwendung innerhalb von verschiedenen Mikroprozessor-Betriebssystemen oder auch die Nutzung des Stacks innerhalb kleinerer Projekte, für deren Implementierung kein Betriebssystem benötigt wird. Innerhalb des Stacks sind nahezu alle Funktionen implementiert, die in dem von der Internet Engineering Taskforce veröffentlichen RFC4294[7] festgelegt wurden. Allerdings setzt µIPv6, wie in *Abbildung 2.3* verdeutlicht, hierbei eine Schicht höher an als die zuvor vorgestellten Stacks, so dass für die Nutzung von µIPv6 für die Datenübertragung zusätzliche Funktionen für den Zugriff auf die Funkschnittstelle benötigt werden.

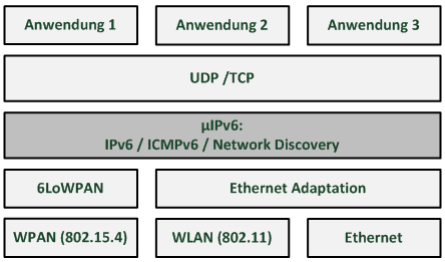

Abb. 2.3: Einordnung von µIPv6

Entwickelt wurde der µIPv6-Stack am Swedish Institute of Computer Science (SICS) unter Leitung von Adam Dunkels, wodurch sich auch die Umbenennung der neueren Versionen in SICSlowpan[8] erklärt. SICSlowpan ist eng mit dem im weiteren Verlauf der Projektdokumentation vorgestellten Mikrocontroller-Betriebssystem ContikiOS verbunden, das ebenfalls von Adam Dunkels entwurfen wurde und welches innerhalb von SICSlowpan zur Prozess- beziehungsweise Threadverwaltung genutzt wird. [8, 11]

2.4 Atmel RUM (Route Under MAC)

Route Under MAC (RUM) ist ein frei verfügbares, von Atmel entwickeltes IEEE 802.15.4 Protokoll, welches in der Lage ist, Pakete bereits auf der MAC-Ebene zu routen. Dies hat insbesondere zum Vorteil, dass Implementierungen von höheren Schichten sich nicht mit dem Routing befassen müssen und der erforderliche Code einfach und schmal gehalten werden kann. Neben den in eine einzelne Schicht ausgelagerten MAC-Funktionen bietet

[6] Weltweites Konsortium, das Leitfäden für den Einsatz IPv6 entwickelt
[7] Request for Comments 4294 - IPv6 Node Requirements
[8] Swedish Institute of Computer Science 6LoWPAN

RUM eine IPv6/6LoWPAN-Schicht, die auf dem zuvor vorgestellten µIPv6-Stack basiert. *Abbildung 2.4* zeigt den Aufbau der einzelnen Schichten des RUM-Pakets. Anzumerken ist hierbei, dass die Programmierung von Kommunikationslösungen mit Hilfe von RUM nicht zwingend die Nutzung aller beinhaltenden Schichten erforderlich macht, sondern auch einzelne Schichten durch den Programmierer neu implementiert oder gegebenenfalls auf diese verzichtet werden kann.

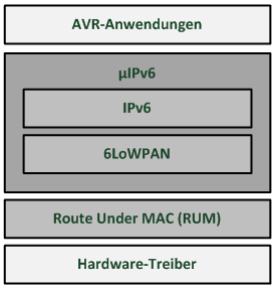

Abb. 2.4: Aufbau des RUM-Pakets

Um den Einstieg in die Programmierung von Anwendungen zu erleichtern, die eine Kommunikation über 6LoWPAN nutzen, wurden dem RUM-Paket von Atmel verschiedene Beispielanwendungen hinzugefügt, die unter anderem die Einsatzmöglichkeiten innerhalb von Sensornetzwerken aufzeigen und durch die Nutzung von Hardware-Abstraktionen zumindest auf nahezu jede Plattform portiert werden können, welche auf Atmel-Mikroprozessoren der AVR-Serie aufbaut. Nachteilig ist jedoch, dass als Basisplattform für die Implementierung des zur Nutzung des RUM-Pakets zwingend notwendigen Koordinators ARM-Prozessoren der SAM7-Serie von Atmel vorgesehen sind. Daher wäre zu diskutieren, ob der mit erhöhten Kosten verbundene Einsatz von ARM-Prozessoren einen Leistungsgewinn für das WieDAS-Projekt erbringen und sich somit rentieren würde. Als Grundlage für den Entschluss wäre unter Umständen in einem weiterführenden Projekt zu untersuchen, welche Auswirkung die Nutzung einer leistungsstärkeren Hardware auf die Übertragungsgeschwindigkeit, die Paketfehlerrate und die maximale Anzahl an Endgeräten hat. [12]

2.5 ContikiOS

Das frei verfügbare, unter BSD-Lizenz stehende, Software-Paket ContikiOS ist ein Betriebssystem für 8-Bit-Computer und Mikrocontroller-Architekturen, in dessen Kategorie zum Beispiel die bereits angesprochenen Mikroprozessoren der AVR-Serie des Herstellers Atmel fallen. Wichtig für den den Einsatz innerhalb des WieDAS-Projekts ist, dass ContikiOS bereits den zuvor beschriebenen µIPv6-Stack und Implementierungen für alle für die Funkkommunkation notwendigen Schichten beeinhaltet.

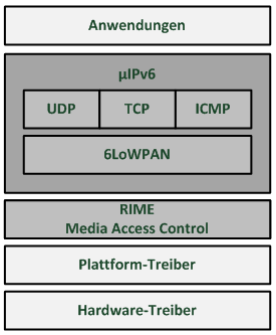

Abb. 2.5: Architektur von ContikiOS

Wie in *Abbildung 2.5* ersichtlich ist, setzt der µIPv6-Kern hierzu auf RIME auf, das neben der reinen Media Access Control (MAC) eine Protokollbibliothek zur Verfügung stellt, die die Umsetzung verschiedener Kommunikationsparadigmen enthält. Die eigentliche Ansteuerung der Kommunikations-

hardware geschieht über Hardware- und Plattformtreiber, die eine relativ gute Möglichkeit für die Portierung auf diverse Hardware-Systeme zu bieten scheint. [8, 13]

Nachteilig wirkt sich bei ContikiOS jedoch das komplexe Build-System[9] aus, das auf die Verwendung des Programmes make in Verbindung mit dem Compiler GCC[10] zugeschnitten ist und neben den Eigenheiten der eigentlichen Programmierung von ContikiOS-Anwendungen weitere Einarbeitungszeit erfordert.

2.6 TinyOS

TinyOS ist ähnlich wie ContikiOS ein größtenteils unter BSD-Lizens veröffentlichtes Betriebssystem, das vornehmlich für den Einsatz in drahtlosen Sensornetzen konzipiert wurde und einen kompletten IPv6-, beziehungsweise 6LoWPAN-Stack, beinhaltet. Ursprünglich wurde TinyOS von einem Team um Dr. David Culler an der University of California (Berkeley, USA) entwickelt, das nach Fertigstellung der ersten Version im Jahr 2000 die Entwicklung jedoch nicht mehr weiter voran trieb. Die Weiterentwicklung geschah danach zunächst vor allem durch die Opensource Gemeinschaft, deren Ergebnisse wiederum in neue Projekte an verschiedenen Forschungseinrichtungen einflossen. Daher ist TinyOS in der Forschungsgemeinde weit verbreitet und bietet im Gegensatz zu ContikiOS eine hervorragende Dokumentation. [14, 15]

Durch die Nutzung von Hardware-Abstraktionen bietet TinyOS grundsätzlich Möglichkeiten zur Portierung auf diverse Hardware-Plattformen. Allerdings hat TinyOS den großen Nachteil, dass die Entwickler für den speziellen Einsatzzweck in Sensorknoten ein eigenes Programmierkonzept und zu dessen Unterstützung einen neuen, von der Programmiersprache *C* abgeleiteten, Dialekt namens *nesC* geschaffen haben, wodurch für die Erstellung von Anwendungen wiederum spezielle Compiler und Hilfsprogramme benötigt werden [15]. Daher wäre für die Nutzung von TinyOS neben Erlernung von *nesC* eine Einarbeitung in diese, durch das TinyOS-Paket bereitgestellten, Programmierwerkzeuge notwendig.

2.7 FreeRTOS

FreeRTOS[11] ist ein Betriebssystem für eingebettete Systeme, das unter einer modifizierten GPL[12]-Lizensierung angeboten wird und sich durch die Modifikation der Lizens auch für den Einsatz in kommerziellen Projekten eignet. Die Kommunikationsfunktionalitäten lassen sich bei der Nutzung von FreeRTOS durch die Einbindung verschiedener TCP/IP-Stacks implementieren, zu denen auch der zuvor schon vorgestellte µIP-Stack zählt. Im

[9] Umgebung für die Softwareerstellung
[10] GNU Compiler Collection
[11] Free Real Time Operation System
[12] GNU General Public License

Gegensatz zu den vorher genannten Betriebsystemen bietet FreeRTOS, wie der Name bereits vermuten lässt, die Möglichkeit echte Realzeit-Anwendungen zu implementieren. Neben der Portierung auf unterschiedlichste Hardware-Plattformen existieren von FreeRTOS einige Ableger, wie zum Beispiel SafeRTOS, das für sicherheitskritische Bereiche konzipiert ist und durch verschiedene Einrichtungen lizensiert wurde. [14, 16]

Die Portierung auf neue Prozessoren und Systeme geschieht innerhalb von FreeRTOS im wesentlichen über zwei Dateien, in denen benötigte Register und Funktionen definiert sind. Nachteilig ist hierbei, dass die Funktionen neben nativen *C* auch in *C++* und zu großen Teilen auf Basis von *Assembler* implementiert wurden. Da nicht jeder Programmierer auf tiefer gehende Erfahrungen in der Assembler-Programmierung zurück greifen kann, kann dies unter Umständen eine Hürde hinsichtlich eigener Portierungen darstellen.

2.8 Für dieses Projekt genutzte Kommunikationslösung

Um eine Entscheidung bezüglich der innerhalb dieses Praxisprojekts zu nutzenden Kommunikationslösing treffen zu können, wurde die in folgender *Tabelle 2.1* dargestellte Entscheidungsmatrix entwickelt. Auf dieser Grundlage wurde zu dem Entschluss gekommen, das Mikrocontroller-Betriebssystem ContikiOS als Basis für die Realisierung der Kommunikation über das 6LoWPAN-Protokoll und die Untersuchung der Möglichkeiten zur Portierung zu nutzen.

	µracoli	**MAC**	**µIPv6**	**RUM**	**Contiki**	**Tiny**	**RTOS**
Offenheit	6	3	6	4	6	6	6
vollst. 6LoWPAN-Stack vorhanden	1	1	2	2	6	6	6
Klarheit der Programmierschnittstellen	6	5	4	4	4	4	4
voraussichtliche Einarbeitungszeit	6	5	4	4	4	3	3
Porgrammierung/Portierung vollständig in C möglich	6	6	6	6	6	1	3
vorhandene Erfahrungen innerhalb des Labors für Informatik	1	1	1	1	3	1	1
Funktionsumpfang	2	2	3	3	5	5	6
Summe	**28**	**23**	**26**	**24**	**34**	**26**	**29**

Tab. 2.1: Entscheidungsmatrix Kommunikationslösung

2.9 Weitere Stacks und Betriebssysteme

Neben den zuvor aufgeführten Möglichkeiten der Implementierung einer Kommunikation über IPv6/6LoWPAN existieren diverse weitere Stacks und Betriebssysteme, die in der Lage sind, Kommunikationsfunktionalitäten über IEEE 802.15.4 bereit zu stellen. Diese sind jedoch häufig für die Verwendung mit leistungsstärkeren Prozesoren als den innerhalb des WieDAS-Projekts genutzten Mikrocontrollern vorgesehen oder nicht unter freien Software-Lizenzen verfügbar. Der Vollständigkeit halber sollen jedoch im Folgenden einige weitere zumindest kurz Erwähnung finden, wobei auf genauere Einlassungen verzichtet wird und kein Anspruch auf Vollständigkeit besteht: [11, 17]

- Arch Rock PhyNet
- NanoStack
- Jennic
- m-Stack
- B6LoWPAN (Berkeley 6LoWPAN)

3 Genutzte Hardware innerhalb des Praxisprojektes

3.1 Das ATmega128RFA1 Evaluation Kit

Für die Durchführung dieses Praxisprojekts wurde das Evaluation Kit ATmega128RFA1-EK1 des Herstellers Atmel genutzt, das zwei Platinen beinhaltet, welche die direkte Nutzung des ATmega128RFA1-Controllers gestatten und für die Verwendung mit dem Atmel STK[1]600 konzipiert wurden. Neben dem Mikrocontroller befinden sich auf der Platine als weitere Hauptbestandteile ein Festspannungsregler, sowie ein Taster und vier LEDs, die eine erste Funktionsüberprüfung des Controllers, beziehungsweise der den Controller steuernden Software, ermöglichen.

Über die Nutzung des STK600 wird zum einen der Anschluss an das entsprechende Programmiergerät, hier das Atmel JTAG ICE mk2, erleichtert und zum anderen die I/O[2]-Ports auf Pfostenstecker geführt, so dass entweder weitere, auf dem STK600 enthaltene Bauteile genutzt oder diverse Elemente, wie zum Beispiel Temperatursensoren oder andere messtechnische Schaltungen angeschlossen werden können. Hinweise zur Nutzung des Evaluation Kits zusammen mit dem STK600 sind in den entsprechenden Unterlagen des Herstellers Atmel [18] zu finden, die zwar die Verwendung eines anderen Mikrocontrollers beschreiben, aber prinzipiell auch als Vorlage für die Vorgehensweise für den Anschluss und die Programmierung des hier genutzten Developement Kits dienen.

Anzumerken ist, dass das STK600 prinzipiell selber als Programmieradapter genutzt werden und somit die Verwendung eines weiteren Programmiergeräts entfallen kann. Allerdings bieten die aktuellen Treiber des STKs noch keine Unterstützung für die Programmierung des innerhalb des Projektes eingesetzten Prozessors ATmega128RFA1, weshalb auf das schon angesprochene Programmiergerät JTAG ICE mk2 zurückgegriffen wird.

[1] Starter Kit
[2] Input/Output

3.2 Kompatibilität des Controllers ATmega128RFA1 zu anderen Prozessoren

Der Mikrocontroller ATmega128RFA1 ist bis auf wenige Veränderungen und der Integration des Funkmoduls identisch zu den Controllern ATmega1281, beziehungsweise ATmega2561. Diese Abwärtskompatibilität, die die Registernamen und -adressen, sowie die Bezeichnungen der durch den Hersteller implementierten Grundfunktionen umfasst, ermöglicht die Nutzung von bereits erstellten Programmen, die für die Verwendung von diversen Prozessoren ausgelegt sind, die auf der ATmega1281-Serie basieren. Allerdings existieren auch einige Inkompatibilitäten, die in den folgenden Abschnitten kurz beschrieben werden sollen. Weitere Informationen können der entsprechenden Kurzanleitung des Herstellers Atmel [19] entnommen werden.

3.2.1 Nicht implementierte Ports

Obwohl die entsprechenden Register verfügbar sind, besitzt der Mikroprozessor ATmega128RFA1 keine Impementierungen für PORT A und PORT C. Dies bedeutet, dass diese Ports, falls sie innerhalb bereits vorhandener Pogramme genutzt werden, auf andere Ports umgelegt werden müssen.

3.2.2 Externer Speicher

Da PORT A und PORT C, die bei den Prozessoren der ATmega1281-Serie als Schnittstelle für die Nutzung von externem Speicher dienen, nicht implementiert wurden, kann der Mikrocontroller ATmega128RFA1 keinen externen Speicher ansprechen. Der Controller verfügt jedoch über einen erweiterten Datenspeicher von 16 kB (SRAM), weshalb davon ausgegangen wird, dass kein externer Speicher und somit auch nicht die dazu gehörige parallele Schnittstelle benötigt werden. Diese Konfiguration trägt dazu bei, die elektromagnetische Abstrahlung gering zu halten und dadurch die Empfindlichkeit einer angeschlossenen Antenne zu erhöhen.

3.2.3 High Voltage Programming

Wegen des Wegfalls von PORT A kann die alternative Funktion des Pins PA0 nicht mehr für die Auswahl des Programmiermodus genutzt werden. Die Auswahl des parallelen Programmiermodus geschieht bei dem Controller ATmega128RFA1 nun über den TST-Pin.

3.2.4 Integrierte und externe Oszilatoren

Der Mikrocontroller ist in der Lage den leistungsfähigen Oszilator des Transceiver-Moduls zu nutzen, welcher an den Eingängen XTAL1 und XTAL2 angeschlossen ist. Außerdem

besteht die Möglichkeit, über den Eingang CLKI, einen externen Taktgeber anzuschließen.

3.2.5 Analoge Ein-/Ausgänge

Der Controller ATmega128RFA1 verfügt über einen neu gestalteten A/D-Wandler, der über die von den Controllern der ATmega1281-Serie bekannten Register angesprochen wird. Allerdings können wegen der höheren Umwandlungsgeschwindigkeit und der größeren Präzision des Wandlers unter Umständen Änderungen an bereits vorhandenen Programmen nötig werden.

3.3 Nutzung der ATmega128RFA1 Transceiver-Einheit

Da der größte Unterschied zwischen dem in diesem Praxisprojekt genutzten Controller ATmega128RFA1 und Prozessoren mit externer Transceiver-Einheit in der direkten Nutzung der Controller-Register zur Steuerung der Datenübertragung liegt, sollen innerhalb dieses Abschnittes die Transceiver-Register und die Grundlagen der Programmierung der drahtlosen Datenübertragung mit Hilfe des Controllers ATmega128RFA1 vorgestellt werden. Tiefer gehende Beschreibungen des Transceiver-Moduls und des gesamten Controllers sind der Anleitung des Mikrocontrollers [20] zu entnehmen. Neben Beschreibungen zur Programmierung werden hier zusätzlich diverse elektrische und funktionelle Eigenschaften des Transceiver-Moduls beschrieben.

3.3.1 Schnittstellen zwischen Transceiver-Einheit und Controller

3.3.1.1 Register-Zugriff

Wie bereits zuvor angedeutet, werden die Transceiver-Register direkt in den Adressraum des Controllers abgebildet, wobei der Zugriff nur möglich ist, wenn das Transceiver-Modul eingeschaltet ist und sich nicht im Sleep-Modus befindet. Da der Zugriff auf die Register zwischen Controller und Transceiver-Einheit asynchron geregelt ist, kann der Register-Zugriff durch die Transceiver-Einheit bis zu vier Taktzyklen des Controllers in Anspruch nehmen. Hierdurch werden in Abhängigkeit von der Taktgeschwindigkeit des Controllers bei wechselseitigem Zugriff auf die Transceiver Register unter Umständen automatisch Wartezyklen generiert. Wird der Controller beispielsweise mit einer Taktgeschwindigkeit von 16 MHz betrieben und das Transceiver soll mehrere Male direkt hintereinander durch den Controller beschrieben und durch den Transceiver ausgelesen werden, muss bei einer Taktung des Transceivers mit 4 MHz mit einer Einfügung von drei Wartezyklen für jeden Lesevorgang der Transceiver-Einheit gerechnet werden. Bei einer Taktung des Controllers mit 4 MHz kann die Einfügung von Wartezyklen entfallen.

3.3.1.2 Zugriff auf den Frame-Buffer

Der Frame-Buffer dient als Puffer zur Abspeicherung der Daten, die über die Funkschnittstelle übertragen werden sollen, beziehungsweise bei eingehenden Paketen über die Funkschnittstelle aufgenommen werden. Da die maximale Größe der Nutzdaten eines nach IEEE 802.15.4 standardisierten Frames 127 Byte beträgt, erlaubt die Größe des Puffers von 128 Byte hierbei die direkte Speicherung eines kompletten Frames. Im Gegensatz zu der direkten Nutzung der Transceiver-Register, erlaubt der Frame-Buffer unabhängig von der genutzten Taktgeschwindigkeit des Controllers die Ausführung von Lese- und Schreiboperationen innerhalb eines Taktzyklus.

Der Frame-Buffer liegt oberhalb der Kontroll-Register des Transceivers innerhalb des Adressraums des Controllers und kann über symbolische Adressen angesprochen werden. Hierzu dienen die symbolischen Namen TRXFBST und TRXFBEND, über die das erste und das letzte Byte des im Frame-Buffer abgelegten Datenrahmens adressiert werden. Der fortlaufende Zugriff auf einzelne Bytes innerhalb des Frames erfolgt über die Nutzung von "TRXFBST + byte index", beziehnungsweise "TRXFBEND - byte index". Die Länge von empfangenen Frames wird jedoch nicht innerhalb des Frame-Buffers abgelegt, sondern über das Register TST_FRAME_LENGTH abgefragt. Sollen Daten gesendet werden, wird die Länge des zu versendenden Frames allerdings nicht über dieses Register übergeben, da beim Versand von Daten die Größe des Frames über das erste Byte des Frames übergeben wird. Anzumerken ist, dass es wegen der unterschiedlichen Nutzung des Frame-Buffers beim Empfang und Versand von Daten nicht möglich ist, einen empfangenen Frame sofort wieder aus dem Puffer zu versenden, ohne ihn vorher zu modifizieren.

3.3.1.3 Das Transceiver Pin Register

Das Transceiver Pin Register TRXPR enthält zwei Konfigurationsbits, über die verschiedene Funktionen der Transceiver-Einheit angesprochen werden. Hierbei wurde der Zugriff auf die Funktionalitäten dem der Lösung der Datenübertragung durch einzelne Controller- und Transceiver-Einheiten, wie zum Beispiel dem AT86RF231 Transceiver-Chip, angepasst. Da das Register innerhalb der Clock Domain des Controllers liegt, ist der Zugriff hierauf auch möglich, wenn sich die Transceiver-Einheit im Sleep-Zustand befindet.

Das Konfigurationsbit TRXRST dient dazu, die Transceiver-Einheit resetten und wird automatisch nach dem Setzen automatisch wieder gelöscht.

Transceiver-Status	Funktion	SLPTR Bit	Beschreibung
PLL_ON	TX start	0 -> 1	Startet die Übertragung eines Frames
TX_ARET_ON	TX start	0 -> 1	Startet eine TX_ARET-Übertragung
TRX_OFF	Sleep	0 -> 1	Versetzt die Transceiver-Einheit in den Schlafmodus
SLEEP	Wakeup	1 -> 0	Versetzt die Transceiver-Einheit in den Status TRX_OFF

Tab. 3.1: Nutzung des SLPTR-Konfigurationsbits

Über das zweite Konfigurationsbit SLPTR können neben der Übertragung von Frames verschiedene Zustände des Transceiver-Moduls gesteuert werden. Hierbei ist zu beachten, dass sich die Funktion je nach aktuellem Zustand der Transceiver-Einheit unterscheidet. *Tabelle 3.1* verdeutlicht die Nutzung des Konfigurationsbits in Abhängigkeit des aktuellen Zustands.

3.3.1.4 Interrupts

Die Transceiver-Einheit unterscheidet zwischen acht verschiedenen Interrupts, die über einzelne Bits innerhalb des Interrupt-Status-Registerers IRQ_STATUS abgefragt werden können. Sollen einer der Interrupts automatisch genutzt werden, ist es zunächst erforderlich, das dem Interrupt zugeordnete Bit innerhalb des Interrupt-Mask-Registers IRQ_MASK zu aktivieren. Zudem muss für jeden aktivierten Interrupt eine Interrupt-Service-Routine definiert werden, die die Behandlung des Interrupts implementiert. Die Rücksetzung des Interrupt-Bits erfolgt automatisch, sobald die entsprechende Interrupt-Service-Routine aufgerufen wird.

Zusätzlich besteht die Möglichkeit, die verschiedenen Interrupts manuell über das Interrupt-Status-Register abzufragen. Da die Interrupt-Bits hierbei nicht automatisch gelöscht werden, wenn die entsprechenden Ereignisse abgearbeitet wurden, muss das zuvor gesetzte Bit manuell durch das Schreiben einer "1" in das jeweilige Register-Bit gelöscht werden.

IRQ-Nr.	IRQ-Bezeichnung	Beschreibung
64	TRX24_AWAKE	Zeigt das Erreichen der Zustände TRX_OFF, RESET oder SLEEP an
63	TRX24_END	Zeigt an, dass ein Frame komplett übertragen wurde
62	TRX24_XAH_AMI	Zeigt die Übereinstimmung von Adressen an
61	TRX24_CCA_ED_DONE	Zeigt das Ende einer CCA- oder ED-Messung an
60	TRX24_RX_END	Zeigt an, dass ein kompletter Frame empfangen wurde
59	TRX24_RX_START	Zeigt den Beginn eines PSDU-Empfangs an
58	TRX24_PLL_UNLOCK	Zeigt die Entsperrung des PLLs an
57	TRX24_PLL_LOCK	Zeigt die Sperrung des PLLs an

Tab. 3.2: Beschreibung der Transceiver-Interrupts

Tabelle 3.2 zeigt die nutzbaren Interrupts der im "Basic Operation Mode" arbeitenden Transceiver-Einheit. Die IRQ[3]-Nummer dient neben der Identifizierung innerhalb des

[3]Interrupt Request

IRQ-Vektors der Priorisierung der IRQs, wobei die niedrigste IRQ-Nummer die höchste Priorität signalisiert.

3.3.1.5 Steuerung des Transceiver-Zustandes

Die Steuerung der verschiedenen Zustände des Transceiver-Moduls geschieht entweder durch Zugriff auf die TRX_CMD-Bits des Registers TRX_STATE oder direkt über die Kontrollbits SLPTR und TRXRST des TRXPR-Registers. Der erfolgreiche Wechsel des Transceiver-Zustandes wird über die Abfrage des Registers TRX_STATUS ermöglicht. Falls dieses Register den Wert "0x1F" (STATE_TRANSITION_IN_PROGRESS) annimmt, bedeutet dies, dass sich der Status der Transceiver-Einheit gerade verändert und deshalb keine weiteren Befehle verwendet werden dürfen, die sich auf den Zustand des Transceivers auswirken.

3.3.1.6 Identifikation der Transceiver-Einheit

Die Transceiver-Einheit des ATmega128RFA1 kann über über die vier Register PART_NUM, VERSION_NUM, MAN_ID_0 und MAN_ID_1 identifiziert werden. Hier sind die einzigartige Bauteilnummer mit der entsprechenden Versionsnummer und zusätzlich die Hersteller-ID der JTAG[4]-Schnittstelle hinterlegt. Je nach Implementierung der IPv6-Übertragung können diese Identifikationsdaten genutzt werden, um die IPv6-Adresse des Geräts zu generieren.

3.3.2 Ablauf der Datenübertragung

Durch die direkte Nutzung der Controller für die Kontrolle der Transceiver-Einheit gestaltet sich die Implementierung der Datenübertragung relativ einfach. In den folgenden Unterabschnitten soll aufgezeigt werden, welche Schritte im einzelnen für die Konfiguration des Transceivers und das Versenden und Empfangen von Datenframes notwendig sind. Der Übersicht halber werden innerhalb der zughörigen Listings lediglich Befehle dargestellt, die der Steuerung der Transceivers dienen.

3.3.2.1 Initialisierung und Konfiguration des Transceiver-Moduls

Bevor Daten empfangen oder gesendet werden können, ist zunächst notwendig, die Transceiver-Einheit zu initialisieren und zu konfigurieren. Hierzu wird der Transceiver zurückgesetzt (Reset) und der Übertragungskanal durch Schreiben in das Subregister PHY_CC_CCA.CHANNEL gesetzt. Neben der Auswahl des Übertragungskanals können je nach Anforderung weitere Übertragunsparameter, wie zum Beispiel die Übertragungsgeschwindigkeit, über verschiedene Register der Transceiver-Einheit eingstellt werden.

[4] Joint Test Action Group

Genauere Beschreibungen zu den Konfigurationsmöglichkeiten sind der Anleitung des Mikrocontrollers [20] zu entnehmen.

Listing 3.1 verdeutlicht eine exemplarische Implementierung der Initialisierung und der Konfiguration des Transceiver-Moduls des ATmega128RFA1.

```
1  [...]
2
3  /* reset transceiver */
4  TRXPR &= ~_BV(TRXRST);
5  TRXPR &= ~_BV(SLPTR);
6  TRXPR |= _BV(TRXRST);
7
8  /* take transceiver to state TRX_OFF */
9  TRX_STATE |= (1 << TRX_CMD3);
10
11 /* setup transmition channel = 16 */
12 PHY_CC_CCA = 0x30;
13
14 [...]
```

Listing 3.1: Initialisierung und Konfiguration des Transceiver-Moduls

3.3.2.2 Empfang von Daten

Für den Empfang von Daten muss die Transceiver-Einheit zunächst in den Status RX_ON oder RX_ACK_ON gesetzt werden. In diesen Betriebsmodi ist das Transceiver-Modul in die Lage, einen Frame über den eingestellten Kanal zu empfangen, zu demodulieren und die empfangenen Daten in den Frame-Buffer zu schreiben. Für das Lesen der Daten aus dem Frame-Buffer bieten sich zwei unterschiedliche Möglichkeiten an. Bei der Implementierung von nicht zeitkritischen Anwendungen wird der Lesezugriff erst gestartet, wenn der Transceiver durch Setzen des TRX24_RX_END-Interrupt-Flags signalisiert hat, dass der komplette Frame empfangen wurde. Eine weitere Implementierungslösung besteht darin, den Lesezugriff bereits bei Beginn des Datenempfangs zu starten, der durch den TRX_24_START-Interrupt angezeigt wird. Diese schnellere Form des Auslesens des Frames wird jedoch daurch erkauft, dass möglicherweise Bytes gelesen werden, die noch nicht mit neuen Daten beschrieben sind. Deshalb sollte letztere Möglichkeit des Lesens nur bei der Realisierung von zeitkritischen Anwendungen genutzt werden.

Das folgende *Listing 3.2* zeigt beispielhaft die für den Empfang eines Datenframes notwendige Befehlsfolge.

```
1  [...]
2
3  /* take transceiver to state RX_ON */
```

```
4  TRX_STATE_CMD = RX_ON;
5
6  /* enable RX_END interrupt */
7  IRQ_MASK |= (1 << RX_END_EN);
8
9  [...]
10
11
12 /* isr forced when receiving of frame is completed */
13 ISR(TRX24_RX_END_vect)
14 {
15   [...]
16
17   memcpy(*data, (void*) &TRXFBST, TST_FRAME_LENGTH);
18 }
```

Listing 3.2: Datenempfang über das Transceiver-Modul des ATmega128RFA1

3.3.2.3 Versenden von Daten

Sollen Daten über das Transceiver-Modul versendet werden, ist es zunächst erforderlich, dieses in den Status PLL_ON oder TX_ARET_ON zu versetzen. Hierauf werden die zu versendenden Daten in den Frame-Buffer geschrieben und die Übermittlung durch Nutzung des Konfigurationsbits SLPTR oder das Setzen des Bits TX_START innerhalb TRX_STATE-Registers gestartet. Durch Abfrage des TX_END-Interrupts wird überprüft, ob der komplette Frame aus dem Frame-Buffer übertragen wurde und, zum Beispiel für einen erneuten Datenversand, wieder auf den Frame-Buffer zugegriffen werden kann. Zusätzlich existiert für die Implementierung von zeitkritischen Anwendungen die Möglichkeit, die Übertragung bereits zu starten, bevor der zu versendende Frame in den Frame-Buffer geschrieben wurde. Hierbei ist darauf zu achten, dass die Zeitspanne zwischen dem Starten der Übertragung und dem Füllen des Frame-Buffers maximal 176 µs beträgt. Dies ist die Zeitspanne die vor der eigentlichen Datenübertragung benötigt wird, um den PLL[5] einzustellen und den durch den Transceiver automatisch generierten Synchronisations-Header (SHR) zu versenden. Zudem ist auf eine schnelle Füllung des Frame-Buffers zu achten, da es ansonsten zu einem “Buffer-Underrun” kommen kann.

Eine mögliche Implementierung des Datenversands ist in *Listing 3.3* dargestellt.

```
1  [...]
2
3  /* take transceiver to state PLL_ON */
4  TRX_STATE_CMD = PLL_ON;
5
6  /* enable TX_END interrupt */
```

[5] Phase-Locked Loop

```
7  IRQ_MASK |= _BV(TX_END_EN);
8
9  /* data to transmit */
10 data = "Testframe";
11
12 /* fill frame-buffer */
13 TRXFBST = sizeof(data);
14 memcpy((void*) &TRXFBST+1, data, sizeof(data));
15
16 /* transmit buffered frame */
17 TRXPR |= _BV(SLPTR)
18 TRXPR &= ~_BV(SLPTR)
19
20 [...]
```

Listing 3.3: Datenversand über das Transceiver-Modul des ATmega128RFA1

3.4 Erste Funktionsüberprüfung

Da für die erste Funktionsüberprüfung mögliche Fehler innerhalb einer eigenen Implementierung der Kommunikationsmechanismen ausgeschlossen werden sollten, wurde hierzu auf das µracoli-Paket zurückgegriffen, mit Hilfe dessen nach einer kurzen Einarbeitung erste Programme geschrieben werden konnten.

Zunächst wurde ein Empfänger implementiert, der über die verschiedenen LEDs des ATmega128RFA1 Developement Kits den momentanen Transceiver-Status und den Empfang eines gültigen Daten-Frames anzeigt. Die Software des Senders wurde so konzipiert, dass bei Betätigung des Tasters des ATmega128RFA1 Developement Kits ein Frame versendet wird, der eine Konstante als Nutzdatenanteil enthält. Auch hier können der aktuelle Status des Transceiver-Moduls und der Versand des Frames über die LEDs des Developement Kits überwacht werden.

Listing A.1 und *Listing A.2*, die im *Anhang A.1* dieser Dokumentation zu finden sind, zeigen die Implementierung der Software des Empfängers, beziehungsweise des Senders.

4 Grundlagen zu ContikiOS

Dieses Kapitel dient der Vermittlung der Grundlagen zu ContikiOS. Ziel ist es, die Einarbeitungszeit für die Erstellung eigener Anwendungen durch den Leser zu verkürzen und die Basis für das Verständnis der Portierung auf neue Hardware-Plattformen zu schaffen. Als Unterstützung hierzu bietet sich die Durchsicht des Kapitels 2 der Bachelorthesis von Herrn Bernhard Esders [3] an, in dem einige Grundlagen zu dem Contiki-Betriebssystem selber und zu der Programmierung von Contiki-Anwendungen aufbereitet wurden.

4.1 Multi-Threading und Protothreads

Neben der optionalen Möglichkeit des präemtiven Multithreadings bietet ContikiOS über die Nutzung von so genannten Protothreads einen leichtgewichtigen Mechanismus für die Realisierung einer quasiparallelen Programm-, beziehungsweise Prozessausführung. Im Gegensatz zu "normalen" Threads besitzen Protothreads keinen eigenen Stapelspeicher (Stack) und sind somit in der Lage, blockierende Kontexte mit äußerst geringem Speicherbedarf abzubilden. Dieses schnelle und speichereffiziente Verfahren ist gerade für die Umsetzung von eingebetteten Systemen interessant, hat jedoch den Nachteil, dass die Inhalte von lokalen Variablen unter Umständen bei einem Kontextwechsel verloren gehen. Sollen die Variableninhalte über einen Kontextwechsel erhalten bleiben, ist deshalb entweder eine globale Deklaration der Variablen oder eine Definition als statisch (`static`) erforderlich. [3, 21]

4.2 Events und Interprozesskommunikation

Die Interprozesskommunikation wird innerhalb des Mikrocontroller-Betriebssystems Contiki über das Senden und Empfangen von Events realisiert. Die Übergabe der Events geschieht hierbei bei dem Kontextwechsel von dem auslösenden Prozess hin zu dem auf das Event wartenden Prozess. [3]

4.3 Verzeichnisstruktur

Die Verzeichnisstruktur des Mikroprozessor-Betriebssystems ContikiOS wurde so aufgebaut, dass eine klare Trennung zwischen den implementierten Anwendungen, den Treibern,

beziehungsweise Hardware-Abstraktionen, und der eigentlichen Logik des Betriebssystems besteht. Dies wird auch durch die in diesem Kapitel vorgestellte Verzeichnisstruktur deutlich. [22, 23]

4.3.1 Das Anwendungsverzeichnis (apps)

Das Anwendungsverzeichnis apps beinhaltet mit Contiki ausgelieferte, hardwareunabhängige, Anwendungen. Werden eigene Anwendungen entwickelt, die später in weiteren Projekten genutzt werden sollen, empfiehlt es sich, diese ebenfalls im Anwendungsverzeichnis abzulegen, da das Build-System von Contiki die Einbindungen von in diesem Verezichnis liegenden Anwendungen stark vereinfacht.

4.3.2 Das Systemverzeichis (core)

Innerhalb des core-Verzeichnisses liegt die eigentliche Logik des Contiki-Betriebssystems. Das core-Verzeichnis ist in mehrere Unterverzeichnisse gegliedert, die verschiedene Funktionsgruppen enthalten.

core/cfs Beinhaltet Funktionen des Contiki-Dateisystems Coffee[1], das für die Implementierung eigener Anwendungen genutzt werden kann.

core/ctk Hier liegen Funktionen des Contiki Tool Kits, das für einige Architekturen die Erstellung grafischer Benutzeroberflächen ermöglicht.

core/dev Enthält Systemdateien zur Ansteuerung verschiedener Peripherie-Funktionen.

core/lib Beinhaltet verschiedene Systemroutinen, zum Beispiel für die Generierung der CRC[2]-Prüfsumme, für die Verwaltung von Ringpuffern oder die Codierung/Decodierung mit Hilfe des Manchester-Codes.

core/loader Dient der Speicherung der Funktionen des Contiki ELF[3]-Loaders, die ein Nachladen von ELF-Objekten im laufenden Betrieb ermöglichen.

[1] Contiki Flash File System
[2] Cycling Redundancy Check
[3] Executable Linkable Format

core/net Enthält die Funktionen zur Realisierung des Datenverkehrs über IP-basierte Netzwerke. Im Hauptverzeichnis befinden sich der µIP-Stack und unterschiedliche IP-Funktionen. Weitere, im folgenden aufgelistete Unterverzeichnisse, bilden verschiedene Schichten des Kommunikationprotokolls ab.

core/net/mac Implementierung der MAC-Schicht

core/net/rime Implementierung der RIME-Bibliothek

core/net/rpl Routingfunktionen

core/sys Beinhaltet Systemfunktionen, wie Realisierung der Timer, der Prozessverwaltung und der Energieüberwachung.

4.3.3 Prozessorspezifischer Code (cpu)

Innerhalb des Verzeichnisses cpu liegen Treiber für verschiedene Mikroprozessoren-Familien, wobei für jede Mikroprozessor-Familie ein eigenes Unterverzeichnis existiert. Da die weitere Unterteilung je nach Prozessortyp variiert, wird im folgenden lediglich auf das der AVR-Serie zugeordnete Verzeichnis cpu/avr eingegangen.

cpu/avr/dev Beinhaltet Definitionen für verschiedene Module von AVR-Prozessoren, wie zum Beispiel Timer oder die Kommunikation über RS232 und USB.

cpu/avr/radio Enthält die Treiber für die Kommunikation über die Funkschnittstelle.

cpu/avr/radio/ieee-manager Interface der MAC-Schicht zu höheren Protokollen

cpu/avr/radio/mac Hilfsfunktionen für die Kontrolle der MAC-Schicht

cpu/avr/radio/rf230 Enthält Treiber für die Kommunikation über das Funkmodul, die hierzu benötigte Registermap[4] und Funktionen für die Bearbeitung von Frames.

cpu/avr/radio/rf230bb Beinhaltet "abgespeckte" Versionen der Treiber für die drahtlose Kommunikation und die hierzu erforderliche Registermap.

[4]Eine Registermap dient der Zuordnung vo symbolischen Namen zu Registeradressen

4.3.4 Dokumentation zu Contiki (doc)

Im Verzeichnis doc sind neben einigen Beispielprogrammen eine auf dem Dokumentationssystem *doxygen* aufbauende Dokumentation zu Contiki abgelegt. Für die Nutzung dieser Dokumentation ist es jedoch notwendig, das *doxygen*-System zu installieren und die Dateien mit Hilfe des *doxygen*-Wizards zu kompilieren. Da die Dokumentation als HTML-Version über den Internetauftritt des Programmierers Adam Dunkels [22] zugänglich ist, wurde innerhalb dieses Praxisprojekts auf eine eigene Kompilierung verzichtet.

Zudem bleibt anzumerken, dass sich die Nutzung der Dokumentation teilweise als problematisch darstellt, da für den Nutzer nicht ersichtlich ist, auf welche Version des Contiki-Betriebssystem sich einzelne Teile der Dokumentation beziehen. Daher erscheint es für die Erstellung eigener Anwendungen eher hilfreich, die mit der jeweiligen Contiki-Version ausgelieferten Beispielprojekte als Grundlage für den Einstieg in die Programmierung eigener Contiki-Anwendungen zu nutzen.

4.3.5 Beispielprojekte (examples)

Innerhalb des examples-Verzeichnisses befinden sich Beipielprojekte der Contiki-Entwickler. Allerdings ist zu beachten, dass einige Projekte nur für die Nutzung mit bestimmten Hardware-Plattformen konzipiert sind.

4.3.6 Plattformspezifischer Code (platform)

Das Verzeichnis platform enthält plattformspezifische Anpassungen von Systemfunktionen und Erweiterungen, die unter anderem die Nutzung bestimmter Peripheriekomponenten ermöglichen. Hierbei existiert für jede Hardware-Plattform ein eigenes Unterverzeichnis. Innerhalb des Unterverzeichnisses avr-raven sind zum Beispiel Anpassungen für das Coffe-Dateisystem oder die Nutzung von SLIP[5] und Erweiterungen, die die Implentierung von Ausgaben über das auf dem Raven Board des Herstellers Atmel enthaltene Display vereinfachen, hinterlegt.

4.3.7 Hilfsprogramme (tools)

Das Verzeichnis tools beinhaltet verschiedene Programme und Dateien, die bei der Erstellung von eigenen Programmen und der Konfiguration und Überwachung von Systemen, die auf Contiki aufbauen, nützlich sein können. Beispiele hierfür sind:

- der Netzwerksimulator *cooja*
- Bibliotheken für die Nutzung von *cygwin*[6]

[5] Serial Line Internet Protocol
[6] POSIX-basierte Umgebung für Windows-Systeme

- Hilfsprogramme für die Erstellung von Coffee-Image Dateien
- und Simulatoren für verschiedene Prozessortypen

4.4 Das Contiki Build-System

Das Contiki Build System wurde so konzipiert, dass die Kompilierung von Anwendungen für verschiedene, bereits innerhalb des Contiki-Betriebssystems implementierte Hardware-Plattformen, möglich ist, ohne den Quellcode der Anwendungen an die entsprechende Hardware anpassen zu müssen. Hierzu ist für die Übersetzung der Contiki-Anwendungen das Programm *make* vorgesehen, das in diesem Kontext alle für die Erstellung des Maschinencodes notwendigen Schritte, wie Übersetzen und Linken, steuert. Die Steuerungsparameter für die Erstellung des Codes werden hierbei über Konfigurationsdateien, den so genannten "Makefiles", angegeben. Deren Verhalten kann wiederum mit Hilfe von Parametern, die gegebenenfalls mit der Eingabe des make-Befehl übergeben werden, beeinflusst werden.

Das Programm *make* ist Bestandteil der geläufigen Linux-Distributionen und aller auf den POSIX[7]-Standard aufbauenden Betriebssysteme, gehört jeoch nicht zu den mit dem Windows-Betriebssystem ausgelieferten Programmen. Daher wird innnerhalb des folgenden Abschnitts explizit auf die verschiedenen Möglichkeiten der Nutzung von *make* unter Windows eingegangen.

4.4.1 Nutzung des Programmes make unter Windows

Einbindung der Makefiles in eine Entwicklungsumgebung

Die Einbindung der Makefiles in eine vorhandene Entwicklungsumgebung, wie zum Beispiel AVR Studio, stellt prinzipiell die beste Möglichkeit dar, Contiki-Anwendungen zu programmieren und zu übersetzen. Neben der Gewöhnung an die Entwicklungsumgebung bietet sich hierbei der Vorteil, dass nicht auf unterstützdende Funktionen, wie das Debugging von Anwendungen oder die Hervorhebung von Befehlen der genutzten Programmiersprache (Syntax Highlighting) verzichtet werden muss. Die Einbindung der Makefiles in die neuste Version des AVR Studios (Version 5.0.1223) war jedoch auf Grund eines Fehlers in der Software nicht möglich. Es bleibt daher abzuwarten, ob der Fehler durch kommende Updates der AVR Studio Software behoben werden wird.

Installation von Linux in eine virtuelle Maschine

Durch die Installation von Linux in eine virtuelle Maschine, die durch verschiedene Programme, wie VM Ware oder Virtual Box bereitgestellt werden kann, ist es möglich,

[7] Portable Operation System Interface

Linux parallel zu Windows zu betreiben. Das Schreiben des Quellcodes der Contiki-Anwendungen geschieht bei Nutzung dieser Variante entweder direkt mit Hilfe von Programmen, die in der virtuellen Maschine gestartet werden, oder es wird auf eine Entwicklungsumgebungen oder Quellcode-Editoren zurückgegriffen, die für das Windows-Betriebssystem entwickelt wurden.

Installation der POSIX-basierten Umgebung *cygwin*

Die Installation von *cygwin* ermöglicht es, innerhalb des Windows-Betriebssystems eine UNIX-artige, dem POSIX-Standard entsprechende Oberfläche aufzurufen, über die das Programm *make* ausgeführt wird. Da von dieser Oberfläche nicht ohne weiteres auf Dateien außerhalb des *cygwin*-Systems zugegriffen werden kann, sollten sich der gesamte Quellcode des Contiki-Betriebssystems und der implementierten Anwendungen in Verzeichnissen befinden, die unterhalb des Hauptverzeichnisses von *cygwin* liegen. Durch das Fehlen einer grafischen Oberfläche innerhalb von *cygwin* ist es sinnvoll, für das Schreiben des Quellcodes eigener Anwendungen Programme zu nutzen, die unter Windows installiert wurden.

Nutzung des GNU make-Pakets unter Windows

Sofern das GNU make-Paket unter Windows installiert wurde, ist es alternativ zu den zuvor vorgestellten Vorgehensweisen möglich, das Programm *make* direkt über die Kommandozeile von Windows aufzurufen. Da das *make*-Programm Bestandteil der GNU Compiler Collection ist, die auch von der Entwicklungsumgebung AVR Studio für die Kompilierung erstellter Anwendungen genutzt wird, kann *make* ohne erneute Installation auf jedem Windows-System verwendet werden, auf dem die Software AVR Studio vorhanden ist. Für die Erstellung des Quellcodes steht es dem Programmierer frei, ob er AVR Studio oder andere Quellcode-Editoren bevorzugt.

4.4.2 Aufbau der Makefiles

Für die Kompilierung von Contiki und Contiki-Anwendungen werden unterschiedliche Makefiles benötigt, die sich in verschiedenen Verzeichnissen befinden. Folgende Übersicht soll dem Leser dabei helfen, den Aufbau und den Inhalt der Makefiles zu verstehen.

Makefile

Das Makefile des Projekts befindet sich innerhalb des Projektverzeichnisses und muss von dem Programmierer an die eigene Anwendung angepasst werden. Es dient als

Einstiegspunkt der Kompilierung und beinhaltet als Minimalangaben das Hauptverzeichnis des Contiki-Betriebssystems und den Ablageort des System-Makefiles von Contiki (Makefile.include). Zusätzlich ist es über dieses Makefile möglich, innerhalb des apps-Verzeichnisses abgespeicherte Anwendungen anzugeben, die bei der Kompilierung dem Contiki-System hinzugefügt werden oder auf externen Quellcode zu verweisen, der zusätzlich zu den Contiki-Anwendungen kompiliert werden soll. *Listing 4.1* verdeutlicht diese grundlegenden Angaben an einem Beispiel.

```
1  # compile broadcast-example project
2  all: broadcast-example
3
4  # include the servreg-hack application
5  APPS = servreg-hack
6
7  # include other sourcecode
8  CONTIKI_SOURCEFILES += example.c
9
10 # specifiy the contiki root directory
11 CONTIKI = ../../..
12
13 # include the contiki-system makefile
14 include $(CONTIKI)/Makefile.include
```

Listing 4.1: Beispiel eines Projekt-Makefiles

Makefile.include

Makfile.include stellt das System-Makefile von Contiki dar, das im Hauptverzeichnis des Contiki-Betriebssystems abgelegt ist. Es enthält die Definitionen des Betriebssystemkerns und fügt bei der Ausführung des *make*-Befehls das plattformspezifische Makefile (Makefile.$(TARGET)) und die Makefiles der im Projekt-Makefile unter APPS angegebenen Anwendungen ein.

Makefile.$(TARGET)

Makefile.$(TARGET) ist im Verzeichnis der entsprechenden Hardware-Plattform, zum Beispiel unter platform/avr-raven, zu finden und stellt das plattformspezifische Makefile des Contiki-Betriebssystems dar. Haptaufgabe dieses Makefiles ist es, über die Variable CONTIKI_TARGET_SOURCEFILES hardwarespezifischen Quellcode zu laden und das Makefile der zu der Plattform passenden CPU einzufügen.

Makefile.$(CPU)

Dieses Makefile enthält Definitionen, die der C-Compiler zur Kompilierung des Contiki-Systems für einen bestimmten Prozessor benötigt. Für die Nutzung verschiedener Compiler

ist es möglich, zusätzliche Anweisungen einzufügen, über welche die zu verwendenden Compiler konfiguriert werden können.

Makefile.$(APP)

In jedem Unterverzeichnis des Anwendungsverzeichnisses apps existiert ein der Anwendung zugehöriges Makefile. Hier lassen sich Regeln für die Übersetzung der Anwedungen bestimmen, die hauptsächlich der Angabe der Quelldateien der Anwendung dienen. *Listing 4.2* zeigt beispielhaft das Anwendungs-Makefile eines Webservers.

```
1 # include process implementations
2 webserver_src = webserver-nogui.c httpd.c http-strings.c
3                  psock.c memb.c httpd-fs.c httpd-cgi.c
4
5 # include several definitions
6 webserver_dsc = webserver-dsc.c
```

Listing 4.2: Beispielhaftes Anwendungs-Makefile

5 Portierung von ContikiOS 2.4

Die Portierung auf neue Hardware-Plattformen wird dadurch erleichtert, dass bei der Entwicklung des Betriebssystems Contiki nach der Premisse vorgegangen wurde, den hardwareabhängigen Code vollständig von dem funktionalen Code des Betriebssystems zu trennen. Bei einer Portierung sind folgende Bereiche betroffen und müssen gegebenenfalls neu implementiert werden: [24]

- die Zeitsteuerung (clock)
- die I/O-Einheiten (Treiber für Netzwerk, Sensoren, serielle Schnittstelle und ähnliches)
- der ELF-Loader
- die Implementierungen zum Multi-Threading
- Anpassungen an die Eigenheiten der Compiler
- die Makefiles

Vereinfacht wird die Portierung, falls bereits Implementierungen innerhalb des Contiki-Betriebssystems für den Prozessor, beziehungsweise die Prozessor-Familie enthalten sind und der Quellcode lediglich an spezifische Eigenheiten der Hardware-Plattform angepasset werden muss. Da der innerhalb dieses Praxisprojekts zum Einsatz kommende Prozessor ATmega128RFA1 des Herstellers Atmel zu anderen Prozessoren und Plattformen der AVR-Serie, wie dem Raven Board, verwandt ist, werden als Basis für die Portierung die entsprechenden, bereits in Contiki implementierten Treiber genutzt.

5.1 Erstellen einer neuen Hardware-Plattform

Um auf Basis einer bereits vorhandenen eine neue Hardware-Plattform zu erstellen, wird das zugehörige Plattform-Verzeichnis (platform/avr-raven) kopiert und die Kopie zum Beispiel in platform/avr-atmega128rfa1 umbenannt. Wichtig ist hierbei, dass der neue Name dieses Verzeichnisses gleichzeitig die Bezeichnung der neuen Plattform innerhalb des gesamten Contiki-Systems darstellt und von nun an unverändert bei der Vollziehung weiterer Schritte übernommen werden muss. Da der Hauptaugenmerk dieses Praxisprojekts auf der Realisierung der Kommunikation über das 6LoWPAN-Protokoll liegt, gehen die nächsten Abschnitte vorwiegend auf Änderungen an den innerhalb des Plattform-Verzeichnisses liegenden Dateien ein, die für die Implementierung der drahtlosen Kommunikation notwendig sind.

Makefile.avr-raven → Makefile.avr-atmega128rfa1

Außer der Anpassung des Dateinamens sind in Bezug auf den Makefile der Hardware-Plattform nur wenige Änderungen notwendig. Wichtig sind hier nur die Anpassung der Mikrocontroller-Bezeichnung und die Abänderung der Dateinamen der durch das Makefile eingefügten Quellcode-Dateien.

contiki-raven-main.c → contiki-rfa1-main.c

Mit Hilfe dieser Datei wird der Zugriff auf verschiedene Parameter, wie die MAC-Adresse oder die PAN-ID[1] geregelt. Zusätzlich ist es möglich, manuell Signaturen zu setzen, die bei der Kompilierung in die ELF-Datei geschrieben und für das spätere Aufspielen des Programms auf den Mikrocontroller benötigt werden. Dies ist jedoch nur notwendig, falls mit älteren Compilern gearbeitet wird, die keine automatische Verarbeitung von Signaturen unterstützen.

contiki-conf.h

Innerhalb der Datei contiki-conf.h sind hauptsächlich Definitionen abgelegt, die das Verhalten der Hardware-Plattform direkt beeinflussen. Dies sind unter anderem die CPU-Geschwindigkeit und damit verbunden die Anzahl der Oszillator-Ticks pro Sekunde oder die maximal erlaubte Anzahl an geöffneten Verbindungen. Da sich die grundlegenden Eigenschaften der Raven Plattform und der des ATmega128RFA1 sehr ähneln, sind in dieser Datei bis auf die Veränderung der Definition der Plattform-Bezeichnung, die der Identifizierung der genutzten Kommunikationshardware innerhalb der Hardware-Abstraktionen dient, keine weiteren Anpassungsarbeiten nötig.

5.2 Anpassen der CPU-spezifischen Hardware-Abstraktionen und Treiber

Die Anpassung der CPU-spezifischen Hardware-Abstraktionen und Treiber geschieht direkt innerhalb der Dateien des Verzeichnisses cpu/avr, beziehungsweise dessen Unterverzeichnissen. Für die Portierung der Kommunkationstreiber ist jedoch zunächst lediglich das Unterverzeichnis cpu/avr/radio/rf230bb von Bedeutung, da hier die von den Treibern benötigte Registermap des Mikrocontrollers und die Treiber für die Bereitstellung der Kommunikationsfähigkeit liegen.

[1] Private Area Network Identification

5.2.1 Die Registermap

Die Registermap ist eine Datei, über die den intern von Contiki genutzten symbolischen Bezeichnungen der Register und Subregister des Mikrocontrollers entweder AVR-Registerbezeichnungen oder direkt die entsprechenden physikalische Adressen zugeordnet werden. Zudem werden innerhalb der Registermap einige Konstanten definiert, welche die Steuerung der Transceiver-Funktionen über die Register vereinfachen. Da sich gerade der Zugriff auf die Register der Transceiver-Einheit des Controllers ATmega128RFA1 von der Steuerung der zuvor verwendeten Transceivern unterscheidet, ist es notwendig, die Registermap zu kopieren, sie umzubenennen und die Zuordnung der innerhalb von Contiki genutzten Registern zu ändern. Folgendes *Listing 5.1* verdeutlicht den Aufbau der Registermap am Beispiel des Registers TRX_STATE.

```
1 /* assign register */
2 #define RG_TX_STATE             TRX_STATE
3
4 /* assign subregister */
5 /* -> address, bitmask, position */
6 #define SR_TRX_CMD              0x142, 0x1f, 0
7
8 /* assign value */
9 #define STATE_TRANSITION        (31)
```

Listing 5.1: Auszug aus der Registermap

5.2.2 Der Kommunikationstreiber

Der innerhalb des Verzeichnisses cpu/avr/radio/rf230bb abgelegte Kommunikationstreiber besteht neben den Registermaps und dem Makefile aus zwei Quellcode- und zwei zugehörigen Header-Dateien. Da die Untersuchung der Kommunikationsmechanismen des Contiki-Betriebssystems naturgemäß in engem Zusammenhang mit diesen Treiberdateien steht, werden innerhalb der folgenden Abschnitte auf diese Dateien eingegangen, obwohl für die Portierung von Contiki auf den ATmega128RFA1 teilweise keine Änderungen notwendig sind.

hal.h

Die Datei hal.h enthält neben einigen Definitionen, die hauptsächlich für die Nutzung externer Transceiver-Einheiten über die SPI-Schnittstelle benötigt werden, verschiedene Prototypen und Makros für den Zugriff auf die Transceiver-Register. Da die Transceiver-Einheit des ATmega128RFA1 direkt über die dazughörigen Register gesteuert wird, kann auf die Änderung der Definitionen für den Zugriff auf die Transceiver-Einheit über die SPI-Schnittstelle verzichtet werden. Allerdings ist es erforderlich, die genutzte Hardware

innerhalb der Header-Datei mit der Bezeichnung aus der in *Kapitel 5.1* bekannt zu machen, da ansonsten beim Compilieren ein Fehler wegen der fehlenden Identifizierung der Hardware-Plattform auftreten würde. Zudem ist es wegen der unterschiedlichen Arbeitsweise der Transceiver-Einheiten des Raven-Boards und des ATmega128RFA1 notwendig, einige Makros neu zu definieren, über die der Zugriff auf das TRXPR-Register gesteuert wird. *Listing 5.2* zeigt die notwendigen Veränderungen an den Makros der Headerdatei.

```
1  #if defined(__AVR_ATmega128RFA1__)
2
3  /* pull the RST pin low */
4  #define hal_set_rst_low()        (TRXPR &= ~(1 << TRXRST))
5
6  /* pull the RST pin high */
7  #define hal_set_rst_high()       (TRXPR |= (1 << TRXRST))
8
9  /* pull the SLP_TR pin high */
10 #define hal_set_slptr_high()     (TRXPR |= (1 << SLPTR))
11
12 /* pull the SLP_TR pin low */
13 #define hal_set_slptr_low()      (TRXPR &= ~(1 << SLPTR))
14
15 /* read current state of the SLP_TR pin */
16 #define hal_get_slptr()          (TRXPR & (1 << SLPTR))
```

Listing 5.2: Pin-Makros innerhalb der Datei hal.h

halbb.c

Innerhalb der Datei halbb.c werden die in der Headerdatei hal.h deklarierten Prototypen implementiert. Da die Anpassung an die Steuerung der Transceiver-Einheit des ATmega128RFA1 bereits innerhalb der Header-Datei geschieht, brauchen hier außer der Einfügung der Registermap des ATmega128RFA1 keine weiteren Änderungen vorgenommen zu werden. Die Dateien hal.h und halbb.c dienen der Abstraktion der Transceiver-Hardware und können somit als Low-Level Treiber betrachtet werden.

rf230bb.h

Die Header-Datei rf230bb.h beinhaltet ebenfalls Definitionen, Makros und Prototypen für die Steuerung der Transceiver-Einheit. Allerdings wird hier eine Schicht höher angesetzt, als bei den Anweisungen innerhalb der Datei hal.h. Dies bedeutet, dass an dieser Stelle nicht mehr direkt auf die Hardware zugegriffen wird, sondern bereits zuvor definierte Funktionen des Low-Level Treibers genutzt werden. Außer der Einfügung der Registermap des ATmega128RFA1 sind daher keine Änderungen innerhalb der Datei rf230bb.h notwendig.

rf230bb.c

Die Implementierung der innerhalb der Headerdatei deklarierten Prototypen findet sich in der Datei rf230bb.c wieder. Es ist nicht nötig, Anpassungen an dieser Datei vorzunehmen.

5.3 Portierung auf eine bisher nicht implementierte Prozessor-Familie

Die Vorgehensweise bei der Portierung von Contiki an eine gänzlich neue Mikroprozessor-Familie gestaltet sich dahingehend schwieriger, dass alle Hardware-Treiber durch den Programmierer neu entwickelt werden müssen. Allerdings beinhaltet das Contiki-Betriebssystem eine native Implementierung (native interface) für den Mikroprozessor und die Hardware-Plattform, die das Grundgerüst der Contiki-Funktionen abbildet und die Portierung auf neue Prozessor-Typen vereinfacht.

5.4 Probleme

Nach der Bearbeitung der obigen Schritte sollte zumindest eine Kompilierung möglich sein und die grundsätzlichen Kommunikationsfunktionen genutzt werden können. Der erste Versuchs der Kompilierung war jedoch durch die Ausgabe von unerwartet vielen Fehlermeldungen gekennzeichnet. Obwohl einige dieser Fehler unterdrückt werden konnten, war eine vollständige Kompilierung des Contiki-Betriebssystems nicht möglich geworden. Als eine Fehlerquelle hat sich hierbei der Zugriff auf die Register und Subregister herausgestellt.

Das erste Problem hierbei ist, dass bei der Adressierung der Register des ATmega128RFA1 die bisher innerhalb der Raven-Plattform genutzte Adressierungsbreite von 8 Bit für die Kontrolle des Transceivers über das SPI-Register nicht mehr ausreicht und es somit zu einer Diskrepanz (Mismatch) zwischen den in der Registermap angegebenen Registeradressen und dem bei der Deklaration der Prototypen vorgesehenden Datentyp uint8_t (8 Bit unsigned Integer) kommt. Dieser Fehler ließe sich durch eine Änderung an der Prototypen-Deklaration und einer Überladung der implementierenden Funktionen beheben. Allerdings müsste hierzu untersucht werden, ob neben den Treibern für die Realsierung der Kommunikation weitere Teile von Contiki hiervon betroffen sind und gegebenenfalls weitere Änderungen vorgenommen werden müssten.

Das größere Problem liegt jedoch in der Realisierung der Funktionen für die Zugriffe auf die Subregister des ATmega128RFA1 über die in der Header-Dateien definierten Makros. Obwohl sich außer der Breite der Registeradressen nichts an der Verarbeitung der Registerzugriffe ändert, wird die Kompilierung mit der Meldung abgebrochen, dass zu wenig Argumente für die Ausführung der Funktion vorhanden seien. Der Grund für diesen Fehler konnte auch nach längerer Recherche nicht gefunden werden.

6 Arbeiten mit ContikiOS 2.5

Im Laufe der dieser Dokumentation zu Grunde liegenden Projektarbeit wurde mit ContikiOS 2.5 eine neue Version des Betriebssystems veröffentlicht. Da aus den bis dahin erstellten Portierungen von Contiki 2.4 nicht innerhalb der hierfür vorgesehenen Zeitspanne das gewünschte Ergebnis einer lauffähigen Kommunikationslösung für den Mikroprozessor ATmega128RFA1 des Herstellers Atmel resultiert war und die neue Version von Contiki bereits eine Implementierung für diesen Prozessor enthielt, wurde entschieden, die Versuche mit Contiki 2.4 zu beenden und die neue Version zu testen.

6.1 Kompatibilität zu älteren Contiki-Versionen

In Hinblick auf die Kompatibilität verschiedener Contiki-Versionen sind für die Weiterführung des WieDAS-Projektes zwei Aspekte von Bedeutung. Zum einen sollten sich bereits erstellte Software-Implementierungen, wie die Software des Wassermelders, möglichst ohne großen Anpassungsaufwand in neue Versionen des Contiki-Betriebssystems integrieren lassen und zum anderen muss eine Kommunikation über das 6LoWPAN-Protokoll zwischen bereits realisierten Geräte mit zukünftigen Neuentwicklungen möglich sein.

6.1.1 Kompatibilität der Befehle

Um die Kompatibilität bereits implementierter Anwendungen zu überprüfen, wurde die Software des Wassermelders zunächst soweit reduziert, dass sie mit Hilfe des Raven Boards getestet werden konnte, ohne weitere Peripherie anschließen zu müssen. Bei dem Versuch der Kompilierung wurde jedoch ersichtlich, dass sich die Befehle für die Realisierung der drahtlosen Kommunikation innerhalb der neuen Contiki-Version geändert hatten und die Anwendung unter Contiki 2.5 nicht mehr lauffähig war.

Dies lässt darauf schließen, dass mit dem Versionwechsel des Contiki-Betriebssystems auch Arbeiten an den Treibermodulen einhergegangen sind. Daraus resultiert, dass bereits implementierte Anwendungen nicht ohne Weiteres auf die neue Contiki-Version übertragen lassen. Allerdings sind zumindest die untersuchten Veränderungen innerhalb der Kommunikationstreiber marginal, so dass der zeitliche Aufwand für die Abänderung des Quelltextes sehr gering ausfallen konnte. *Listing A.3*, das im *Anhang A.1* dieser Projektdokumentation zu finden ist, zeigt die Implementierung eines Moduls, das nach einer vorgegebenen Zeitspanne ein IPv6-Broadcast-Paket sendet.

6.1.2 Kompatibilität des Kommunikationsprotokolls

Obwohl durch die Messung der Stromaufnahme des ATmega128RFA1 Developement Kits verifiziert wurde, dass das Funkmodul des ATmega128RFA1 arbeitete, konnte zunächst kein Datenempfang auf der Empfängerseite, die über einen Raven USB-Stick mit aufgespielten Contiki in der Version 2.4 realisiert wurde, festgestellt werden. Daher wurde auf den Raven USB-Stick die im Betriebssystem Contiki 2.5 enthaltende Empfangssoftware installiert und der Test erneut durchgeführt. Durch diese Maßnahme wurde die Kommunikation zwischen Sender und Empfänger ermöglicht und die versendeten Broadcast-Pakete ließen sich mit Hilfe des Netzwerk-Sniffer WireShark anzeigen. Daher scheint es so, dass die Änderung der Kommunikationstreiber des Contiki-Betriebssystems eine Veränderung in der Übertragung bewirkt haben, die eine gleichzeitige Nutzung der verschiedenen Contiki-Versionen nicht erlauben.

6.2 Tests zur Datenübertragungsgeschwindigkeit

Da die Verwendung des Mikrocontrollers ATmega128RFA1 wegen der direkten Nutzung der Register zur Steuerung der Transceiver-Einheit unter anderem den Vorteil einer höheren Datenübertragungsgeschwindigkeit bieten könnte, wurden zur Verifizierung dieser Annahme vor dem Ende des Projekts erste Vergleichsmessungen durchgeführt. Hierzu wurden für das ATmega128RFA1 Developement Kit und für das Raven Board geänderte Versionen der Software zum Versand von IPv6-Broadcast-Paketen erstellt, die das Funkmodul veranlasst, die Pakete mit der maximal möglichen Wiederholrate zu senden.

Abb. 6.1: Vergleich der Datenübertragungsraten (brutto)

In *Abbildung 6.1* sind die hierbei gemessenen Datenübertragungsraten (butto) in Abhängigkeit von der Größe der Nutzdaten erichtlich, wobei mehrere Punkte auffallend sind:

1. Der Datendurchsatz des ATmega128RFA1 ist bis zu 50% größer als der des Raven Boards.
2. Der Datendurchsatz des ATmega128RFA1 erreicht fast den für die Kommunikation über den Standard IEEE 802.15.4 vorgesehenen maximalen Durchsatz von 250 kBit/s.
3. Die Datenübertragungsrate beider Systeme erleidet bei der Übertragungs eines Nutzdatengröße von 720 Bit (90 Byte) einen starken Einbruch. Hier liegt die Vermutung nahe, dass nach dem Hinzufügen der Verwaltungsdaten die maximale, innerhalb des Standards IEEE 802.15.4 definierte Framegröße von 1016 Bits (127 Byte) überschritten wird und der Datendurchsatz durch die dadurch notwendige Segmentierung verschlechtert wird.

Weitere Details sind den Messprotokollen zu entnehmen, die im *Anhang A.2* als *Tabellen A.1* und *A.2* hinzugefügt wurden.

7 Fazit / Ausblick

Für den Einstieg in dieses im Labor für Informatik an der Fachhochschule Düsseldorf durchgeführte Projekt war zunächst ein hoher zeitlicher Aufwand für die Aneignung von theoretischen Grundlagen und insbesondere für die Untersuchung verschiedener Kommunikationslösungen nötig. Der Vergleich dieser Kommunikationslösungen resultierte in dem Ergebnis, dass vielfältige Möglichkeiten für die Realisierung der Kommunkation über den Standard IEEE 802.15.4 existieren.

Von diesen Möglichkeiten scheint jedoch in der Summe der Eigenschaften das bereits innerhalb einer früheren Projektarbeit im Labor für Informatik genutzte Mikrocontroller-Betriebssystem Contiki die beste Lösung für den Einsatz innerhalb des WieDAS-Projekts zu sein. Dies liegt vor allem daran, dass Contiki einen vollständigen 6LoWPAN-Stack enthält, der durch die vorhandenen Portierungen des Contiki-Betriebssystems die Implementierung von Kommunikationslösungen für eine große Anzahl an unterschiedlichen Mikrocontrollern ermöglicht. Zudem besitzt Contiki einen geringen Ressourcenbedarf, wodurch das Betriebssystem besonders interessant für den Einsatz bei der Realisierung von eingebetteten Systemen ist.

Allerdings sind während der Durchführung des Praxisprojekts auch einige Schwächen des Contiki-Betriebssystems aufgefallen. Die größte Schwäche ist die Dokumentation zu Contiki, deren Nutzen für den Entwickler gerade nach größeren Entwicklungssprüngen, wie nach der Veröffentlichung einer neuen Version, recht gering ist, was daran liegt, dass Teile der Dokumentation anscheinend nicht richtig gepflegt werden. Zudem wurde mit der Veröffentlichung der neuen Contiki-Version die Internetseite des Betriebssystems neu designed und viele Verlinkungen auf wichtige Dokumente entfernt. Daher waren einige Teile der Dokumentation nach der Umstellung nur schwer aufzufinden. Zusammen mit der Komplexität des Build-Systems und der Verschachtelung der verschiedenen Funktionen untereinander erschwert dies den Einstieg in die Programmierung von Contiki-Anwendungen und die Realierung von komplexeren Projekten, wie die Portierung des Betriebssystems. Eine zweite Schwäche ist die Inkompatibilität der verschiedenen Contiki-Versionen untereinander. Hier ist insbesondere interessant, dass trotz Nutzung des gleichen Kommunikationsprotokolls kein Datenaustausch zwischen Geräten möglich ist, die verschiedene Contiki-Versionen verwenden. Da jedoch gerade die Mechanismen für die Realisierung der drahtlosen Kommunikation prinzipiell den größten Vorteil des Contiki-Betriebssystems darstellen, sollte der gesamte Kommunikationsmechanismus noch einmal genauer untersucht werden, um mögliche Konfigurationsfehler ausschließen zu können. Diese Überprüfung der Implementierung des Kommunikations-Stacks war auf Grund von Zeitmangel zu Ende des Projekts nicht mehr möglich.

Hardwareseitig bleibt festzuhalten, dass der Mikrocontroller ATmega128RFA1 während der Durchführung des Projekts durch die Vorteile des integrierten Funkmoduls überzeugen konnte. Die Datenübertragung über die Transceiver-Einheit des Controllers funktioniert reibungslos und ist wie bereits vermutet wurde, durch den Wegfall der bei der Nutzung von externen Transceivern notwendigen Kommunikation über die SPI-Schnittstelle, performanter.

Für nachfolgende und auf diesem Praxisprojekt aufbauende Arbeiten bietet es sich an, die bisher erarbeiteten Grundlagen weiter zu vertiefen und die Kommunikation zwischen den innerhalb des AAL-Systems genutzten Komponenten zu optimieren. Ansatzpunkte könnten hier die Erhöhung der Performance und die Verbesserung des Energiebedarfs der mobilen Geräte durch den Austausch der bisher genutzten Lösung, die auf der Verwendung des Mikrocontrollers ATmega1284P und des externen Transceivers AT86RF230 basieren, gegen den innerhalb dieses Projekts verwendeten ATmega128RFA1-Chip sein. Weitere Tätigkeitsfelder sind in der Erforschung erweiterter Kommunikationsmechanismen, wie Routing, Verschlüsselung oder der Realisierung von verbindungsorientierten Protokollen zu sehen.

Zudem bleibt zu diskutieren, ob im Rahmen eines längerfristigen Projekts und unter Umständen aufbauend auf bereits vorhandenen Lösungen eine eigene Implementierung des 6LoWPAN-Stacks als sinnvoll erscheinen könnte.

A Anhang

A.1 Listings

```
1  /*************************************
2   copyright (c) 2007 Axel Wachtler
3   modified by Mirco Kern, FH Duesseldorf
4  *************************************/
5
6  /* example: receiving frames */
7  /*****************************/
8
9  /* include header files */
10 #include "board.h"
11 #include "transceiver.h"
12 #include "ioutil.h"
13 #include  <util/crc16.h>
14 #include "xmpl.h"
15
16
17 /* variable storing received frame*/
18 static uint8_t rxfrm[MAX_FRAME_SIZE];
19
20 int main(void)
21 {
22   /* initialize leds (red switched on) */
23   LED_INIT();
24
25   /* initialize the transceiver */
26   DELAY_US(TRX_INIT_TIME_US);
27   TRX_RESET_LOW();
28   TRX_SLPTR_LOW();
29   DELAY_US(TRX_RESET_TIME_US);
30   TRX_RESET_HIGH();
31   trx_reg_write(RG_TRX_STATE,CMD_TRX_OFF);
32   DELAY_US(TRX_INIT_TIME_US);
33
34   /* setup transmitter
35    * - configure radio channel
36    * - go into rx state,
37    * - enable "receive end" irq
38    */
```

```
39    trx_bit_write(SR_CHANNEL,CHANNEL);
40    trx_reg_write(RG_TRX_STATE,CMD_RX_ON);
41    trx_reg_write(RG_IRQ_MASK,TRX_IRQ_RX_END);
42
43    /* enable global interrupts */
44    sei();
45
46    /* wait 500 ms and indicate ready state */
47    WAIT500MS();
48    LED_SET_VALUE(1);
49
50    /* loop and wait for frame receiving */
51    while(1);
52  }
53
54
55  /* isr forced when receiving of frame is completed */
56  ISR(TRX24_RX_END_vect)
57  {
58    /* variable storing frame length */
59    uint8_t flen;
60    /* pointer to received frame */
61    uint8_t *pfrm;
62    /* variable storing result of checksum comparison */
63    uint16_t crc;
64
65    /* upload frame and check for crc validity */
66    pfrm = rxfrm;
67    flen = trx_frame_read(pfrm, sizeof(rxfrm), NULL);
68    crc = 0;
69    do
70    {
71        crc = _crc_ccitt_update(crc, *pfrm++);
72    }
73    while(flen--);
74
75    /* toggle red led if crc is correct */
76    if (crc == 0)
77    {
78      LED_TOGGLE(2);
79    }
80
81    /* wait 500 ms and indicate ready state */
82    WAIT500MS();
83    LED_TOGGLE(1);
84    LED_TOGGLE(0);
85  }
```

Listing A.1: Datenempfang mit Hilfe des µracoli-Pakets

```
1  /*************************************
2   copyright (c) 2007 Axel Wachtler
3   modfied by Mirco Kern, FH Duesseldorf
4  *************************************/
5
6
7  /* example: transmitting frames */
8  /*******************************/
9
10 /* include header files */
11 #include "board.h"
12 #include "transceiver.h"
13 #include "ioutil.h"
14 #include "xmpl.h"
15 #include <util/delay.h>
16
17
18 /* variable indicating tx state */
19 static volatile bool tx_in_progress;
20
21 /* variable indicating button state */
22 static volatile bool button_pressed;
23
24
25 int main(void)
26 {
27   /* define PE5 as input */
28   PORTE |= _BV(PE5);
29   /* setup pull ups */
30   MCUCR = 0x00;
31   /* configure interrupt registers */
32   EIMSK |= _BV(INT5);
33
34   /* ieee 802.15.4 frame */
35   uint8_t txfrm[] =
36     {1,0, /* faked ieee 802.15.4 data frame control field */
37      01,  /* sequence counter */
38      '1','2','8','R','F','A','1',':',' ','H','e','l','l','o',
39      'X','X' /* crc bytes (overwritten by transceiver */
40     };
41
42   /* initialize leds (red switched on) */
43   LED_INIT();
44
45   /* initialize the transceiver */
46   DELAY_US(TRX_INIT_TIME_US);
47   TRX_RESET_LOW();
48   TRX_SLPTR_LOW();
```

```
49    DELAY_US(TRX_RESET_TIME_US);
50    TRX_RESET_HIGH();
51    trx_reg_write(RG_TRX_STATE,CMD_TRX_OFF);
52    DELAY_US(TRX_INIT_TIME_US);
53
54    /* setup transmitter
55     * - configure radio channel
56     * - enable transmitters automatic crc16 generation
57     * - go into TX state,
58     * - enable "transmit end" IRQ
59     */
60    trx_bit_write(SR_CHANNEL,CHANNEL);
61    trx_bit_write(SR_TX_AUTO_CRC_ON,1);
62    trx_reg_write(RG_TRX_STATE,CMD_PLL_ON);
63    trx_reg_write(RG_IRQ_MASK,TRX_IRQ_TX_END);
64
65    /* enable global interrupts */
66    sei();
67
68    /* wait 500 ms and indicate ready state */
69    WAIT500MS();
70    LED_SET_VALUE(1);
71
72    /* initialize variables */
73    tx_in_progress = false;
74    button_pressed = false;
75
76    while(1)
77    {
78      if (button_pressed == true)
79      {
80        /* indicate busy state */
81        LED_TOGGLE(1);
82        LED_TOGGLE(0);
83
84        /* send frame when button is pressed */
85        if (tx_in_progress == false)
86        {
87          tx_in_progress = true;
88          trx_frame_write (sizeof(txfrm), txfrm);
89          TRX_SLPTR_HIGH();
90          TRX_SLPTR_LOW();
91
92            /* "debounce" button */
93                  _delay_ms(100);
94          button_pressed = false;
95        }
96      }
97    }
```

```
98  }
99
100
101 /* isr forced when transmission of frame is completed */
102 ISR(TRX24_TX_END_vect)
103 {
104   /* transmission completed */
105   tx_in_progress = false;
106
107   /* indicate transmission of frame */
108   LED_TOGGLE(2);
109
110   /* wait 500 ms and indicate ready state */
111   WAIT500MS();
112   LED_TOGGLE(0);
113   LED_TOGGLE(1);
114 }
115
116
117 /* isr forced when button is pressed */
118 ISR (INT5_vect)
119 {
120   /* indicate button state */
121   button_pressed = true;
122 }
```

Listing A.2: Datenversand mit Hilfe des µracoli-Pakets

```
1  /*************************************************************
2   Copyright (c) 2011, Swedish Institute of Computer Science.
3   All rights reserved.
4
5   Modified by Mirco Kern, FH Duesseldorf
6  *************************************************************/
7
8
9  /* example: sending broadcasts */
10 /*******************************/
11
12 /* include header files */
13 #include "contiki.h"
14 #include "sys/etimer.h"
15 #include "net/uip.h"
16 #include "net/uip-ds6.h"
17 #include "simple-udp.h"
18 #include <stdio.h>
19 #include <string.h>
20
21 /* define destination port */
22 #define UDP_PORT 4321
23
24 /* define send interval */
25 #define SEND_INTERVAL           (20 * CLOCK_SECOND)
26
27 /* struct storing connection */
28 static struct simple_udp_connection broadcast_connection;
29
30 /*---------------------------------------------------------------*/
31 /* declarate process */
32 PROCESS(broadcast_example_process, "UDP broadcast example process");
33
34 /* start process automaticly */
35 AUTOSTART_PROCESSES(&broadcast_example_process);
36
37 /*---------------------------------------------------------------*/
38 /* storing connection data */
39 static void
40 receiver(struct simple_udp_connection *c,
41          const uip_ipaddr_t *sender_addr,
42          uint16_t sender_port,
43          const uip_ipaddr_t *receiver_addr,
44          uint16_t receiver_port,
45          const uint8_t *data,
46          uint16_t datalen)
47
48 /*---------------------------------------------------------------*/
```

```
49 /* broadcast process */
50 PROCESS_THREAD(broadcast_example_process, ev, data)
51 {
52   /* timer struct */
53   static struct etimer send_timer;
54   /* address variable */
55   uip_ipaddr_t addr;
56
57   /* process starts here */
58   PROCESS_BEGIN();
59
60   /* register udp connection */
61   simple_udp_register(&broadcast_connection, UDP_PORT,
62                       NULL, UDP_PORT,
63                       receiver);
64
65   /* loop */
66   while(1) {
67     /* set timer and wait until timer is expired */
68     etimer_set(&send_timer, SEND_TIME);
69     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&send_timer));
70
71         /* create multicast link */
72     uip_create_linklocal_allnodes_mcast(&addr);
73
74         /* send string */
75     simple_udp_sendto(&broadcast_connection,
76                           "Atmega128RFA1_Broadcast", 23, &addr);
77   }
78
79   /* process ends here*/
80   PROCESS_END();
81 }
82 /*---------------------------------------------------------------------*/
```

Listing A.3: Broadcast-Sender unter Contiki 2.5

A.2 Protokolle der Geschwindigkeitstests

Anzahl Zeichen	Paketgröße (in Bit)	Payload (in Bit)	Anzahl Pakete (in 1/Min.)	Datendurchsatz brutto (in kBit/s)	Datendurchsatz netto (in kBit/s)
10	576	80	15902	152,66	21,20
20	656	160	14501	158,54	38,67
30	769	240	13347	163,72	53,39
40	816	320	12340	167,82	65,81
50	896	400	11486	171,52	76,57
60	976	480	10794	174,85	85,99
70	1056	560	10072	177,27	94,01
80	1136	640	9509	180,04	101,43
90	1216	720	8987	182,14	107,84
100	1296	800	5491	118,61	73,21
200	2096	1600	3345	116,86	89,20
400	3696	3200	1877	115,62	100,11
600	5296	4800	1305	115,19	104,40
800	6896	6400	1000	114,93	106,67
1000	8469	8000	810	114,70	108,00
1200	10096	9600	682	114,76	109,12
1232	10352	9856	672	115,94	110,39

Tab. A.1: Datendurchsatz Raven Board → Raven USB

Anzahl Zeichen	Paketgröße (in Bit)	Payload (in Bit)	Anzahl Pakete (in 1/Min.)	Datendurchsatz brutto (in kBit/s)	Datendurchsatz netto (in kBit/s)
10	576	80	24808	238,16	33,08
20	656	160	21566	235,79	57,51
30	769	240	19086	234,12	76,34
40	816	320	17103	232,60	91,22
50	896	400	15496	231,41	103,31
60	976	480	14173	230,55	113,38
70	1056	560	13053	229,73	121,83
80	1136	640	12099	229,07	129,06
90	1216	720	11272	228,45	135,26
100	1296	800	7298	157,64	97,31
200	2096	1600	4321	150,95	115,23
400	3696	3200	2381	146,67	126,99
600	5296	4800	1643	145,02	131,44
800	6896	6400	1255	144,24	133,87
1000	8469	8000	1015	143,72	135,33
1200	10096	9600	852	143,36	136,32
1232	10352	9856	837	144,41	137,49

Tab. A.2: Datendurchsatz ATmega128RFA1 → Raven USB

Abbildungsverzeichnis

1.1 OSI-Referenzmodell, TCP/IP-Referenzmodell und der 6LoWPAN-Stack im Vergleich 4

2.1 Struktur des µracoli-Pakets 6
2.2 Architektur des MAC-Stacks 7
2.3 Einordnung von µIPv6 8
2.4 Aufbau des RUM-Pakets 9
2.5 Architektur von ContikiOS 9

6.1 Vergleich der Datenübertragungsraten (brutto) 36

Tabellenverzeichnis

2.1 Entscheidungsmatrix Kommunikationslösung 11

3.1 Nutzung des SLPTR-Konfigurationsbits 16
3.2 Beschreibung der Transceiver-Interrupts 17

A.1 Datendurchsatz Raven Board → Raven USB 47
A.2 Datendurchsatz ATmega128RFA1 → Raven USB 48

Listingverzeichnis

3.1 Initialisierung und Konfiguration des Transceiver-Moduls 19
3.2 Datenempfang über das Transceiver-Modul des ATmega128RFA1 19
3.3 Datenversand über das Transceiver-Modul des ATmega128RFA1 20

4.1 Beispiel eines Projekt-Makefiles 28
4.2 Beispielhaftes Anwendungs-Makefile 29

5.1 Auszug aus der Registermap 32
5.2 Pin-Makros innerhalb der Datei hal.h 33

A.1 Datenempfang mit Hilfe des µracoli-Pakets 40
A.2 Datenversand mit Hilfe des µracoli-Pakets 42
A.3 Broadcast-Sender unter Contiki 2.5 45

Literaturverzeichnis

[1] WieDAS-Projekt Team, WieDAS-Projekt Website - Über uns (2011).
URL http://www.wiedas.org/index.php/wiedas

[2] Atmel Corporation, Application Note: AVR2016: RZRAVEN Hardware User's Guide (April 2008).
URL http://www.atmel.com/dyn/resources/prod_documents/8117.pdf

[3] Bernhard Esders, Fachhochschule Düsseldorf, Bachelor-Thesis: Entwicklung, Implementierung und Evaluierung einer speichereffizienten, drahtlosen Kommunikation von eingebetteten Systemen mithilfe des IP-Protokolls basierend auf einem Multitasking-Betriebssystem für 8-Bit AVR Mikrocontroller (August 2010).

[4] Wikipedia, IPv6 (November 2011).
URL http://de.wikipedia.org/wiki/IPv6

[5] teltarif.de Onlineverlag GmbH, IPv6 - Das Internet-Protokoll der nächsten Generation.
URL http://www.teltarif.de/internet/ipv6/

[6] teltarif.de Onlineverlag GmbH, Datenschutz und IPv6: So anonym surfen Sie wirklich.
URL http://www.teltarif.de/internet/ipv6/datenschutz.html

[7] Stefan Hartl, IEEE 802.15.4, ZigBee und 6LoWPAN.
URL https://www.auto.tuwien.ac.at/LVA/HGA/stp/hartl.pdf

[8] Shelby Zach & Carsten Bormann, 6LoWPAN - The Wireless Embedded Internet, John Wiley & Sons, 2009.

[9] μracoli-Projekt Team, μracoli Manual (Januar 2010).
URL http://www.nongnu.org/uracoli/docs.html

[10] Atmel Corporation, Application Note AVR2025: IEEE 802.15.4 MAC Software Package - User Guide (November 2011).
URL http://www.atmel.com/dyn/resources/prod_documents/doc8412.pdf

[11] Usman Sarwar u. a., A Comparative Study on Available IPv6 Platforms for Wireless Sensor Network (2010).
URL http://www.waset.org/journals/waset/v62/v62-164.pdf

[12] Atmel Corporation, Application Note AVR2070: Route Under MAC (RUM) with IPv6 and 6LoWPAN (Juni 2009).
URL http://www.atmel.com/dyn/resources/prod_documents/doc8240.pdf

[13] Dr.-Ing Gerald Kupris & Prof. Dr.-Ing. Axel Sikora, ZigBee - Datenfunk mit IEEE 802.15.4 und ZigBee, Francis Verlag GmbH, 2007.

[14] Jean-Philippe Vasseur & Adam Dunkels, Interconnecting Smart Objects With IP, Elsevier Inc., 2010.

[15] Wikipedia, TinyOS (Mai 2010).
URL http://de.wikipedia.org/wiki/TinyOS

[16] Wikipedia, FreeRTOS (Oktober 2011).
URL http://de.wikipedia.org/wiki/FreeRTOS

[17] Chen Yibo, Haiying Zhou u. A., 6LoWPAN Stacks: A Survey (2011).
URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6040344

[18] Atmel Corporation, Application Note AVR1900: Getting started with ATxmega128A1 on STK600 (April 2008).
URL http://www.atmel.com/dyn/resources/prod_documents/doc8107.pdf

[19] Atmel Corporation, ATmega128RFA1 Preliminary Summary, Revision C (Oktober 2011).
URL http://www.atmel.com/dyn/resources/prod_documents/8266S.pdf

[20] Atmel Corporation, ATmega128RFA1 Preliminary, Revision C (Oktober 2011).
URL http://www.atmel.com/dyn/resources/prod_documents/doc8266.pdf

[21] Wikipedia, Protothreads (September 2011).
URL http://de.wikipedia.org/wiki/Protothread

[22] Adam Dunkels, Contiki Source Code Documentation (Oktober 2009).
URL http://sics.se/~adam/contiki/docs/

[23] Adam Dunkels, Contiki Crash Course (März 2007).
URL http://sics.se/~adam/contiki-workshop-2007/workshop07programming.ppt

[24] Adam Dunkels, Porting Contiki - Crash Course (März 2007).
URL http://sics.se/~adam/contiki-workshop-2007/workshop07porting.ppt

Selbständigkeitserklärung

Hiermit erkläre ich an Eides statt, dass ich die am heutigen Tage vorgelegte Dokumentation der Projektarbeit zum Thema

Untersuchung eines Mikroprozessor-Betriebssystems
mit 6LoWPAN-Stack in Hinblick auf die Portierung auf neue Prozessoren

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Düsseldorf, den 07.01.2012

Unterschrift